

USENIX

CONFERENCE
PROCEEDINGS

Summer
1986

Atlanta,
Georgia





USENIX Association

Summer Conference Proceedings

Atlanta 1986

June 9 - 13, 1986
Atlanta, Georgia USA

For additional copies of these proceedings, write:

USENIX Association

P. O. Box 7

El Cerrito, CA 94530 USA

Price: \$25.00 plus \$25.00 for overseas mail

© Copyright 1986 by The USENIX Association

All rights reserved.

This volume is published as a collective work.

Rights to individual papers remain
with the author or the author's employer.

UNIX® is a registered trademark of AT&T.

Other trademarks are noted in the text.

ACKNOWLEDGEMENTS

Sponsor:	The USENIX Association P. O. Box 7 El Cerrito, CA 94350	
Program Chairperson:	Mike O'Dell	
Program Committee:	John Chambers Mike Hawley Sam Leffler Jim McKie Dennis Ritchie Spencer Thomas	MCC Lucasfilm, Ltd. Lucasfilm, Ltd. Bell Communications Research AT&T Bell Laboratories University of Utah, Computer Science Department
Tutorial Coordinator:	Michael Tilson	Human Computing Resources Corp.
USENIX Meeting Planner:	Judith DesHarnais	
Vendor Exhibition Manager:	John Donnelly	
Conference Hosts:	Paul Manno Dan Forsyth	Medical Systems Development Corp.

Terry Countryman, Peter Wan, and Lillie Elliot of Medical Systems Development Corporation assisted in the production of these proceedings. Brent Laminack of In Touch Ministries graciously provided assistance in producing the table of contents. Ralph Amiel of Standard Press patiently answered many dumb questions on the production of a volume this size.

To all of the authors: You deserve many thanks and much appreciation. You have made an otherwise impossible task relatively easy by providing the camera-readies in plenty of time to meet the printing deadlines. To those authors who forgot to clean their laser printers before generating their copy: Yes, it does show. What you see is what the camera sees. Please remember next time.

TABLE OF CONTENTS

PLENARY SESSION

Wednesday, June 11, 9:00 - 10:30

(Grand Ballroom)

Opening Remarks:

Conference Organizers and the USENIX Board

Keynote Address: Pictures of Programs

Jon Bentley, AT&T Bell Laboratories

MUSIC

Wednesday, June 11, 11:00 - 12:30

(Grand Ballroom)

MIDI Music Software for UNIX	1
<i>Michael Hawley, The Droid Works</i>	
(201) 644-2332 or Eedie & Eddie on the Wire: An Experiment in Music Generation	13
<i>Peter S. Langston, Bell Communications Research</i>	

Networks I

Wednesday, June 11, 2:00 - 3:30

(Grand Ballroom)

Secure Networking in the Sun Environment	28
<i>Bradley Taylor, David Goldberg, Sun Microsystems, Inc.</i>	
A Framework for Networking in System V	38
<i>David J. Olander, Gilbert J. McGrath, Robert K. Israel, AT&T</i>	
OSI and TCP/IP Protocols on a UNIX System V	46
<i>Jean Marc Fenart, Marc Fievet, Christian Huitema, Bernard Martin, Annie Remille, Guy Vaysseix, INRIA</i>	

Performance

Wednesday, June 11, 2:00 - 3:30

(Grand Salon)

Managing Development of Performance-Constrained UNIX-Based Software System on Microcomputers <i>L. Perkins, Martin Marietta</i>	
A Multiuser Multiprocessor Benchmark to Compare UNIX Systems	59
<i>Philip M. Mills, NCR Corporation</i>	
A System Call Tracer for UNIX	72
<i>R. Rodriguez, Digital Equipment Corporation</i>	

Operating Systems 1

Wednesday, June 11, 4:00 - 5:30

(Grand Ballroom)

A New Virtual Memory Implementation for UNIX	81
<i>Edward W. Sznyter, Patrick Clancy, James Crossland, Tektronix, Inc.</i>	
Mach: A New Kernel Foundation for UNIX Development	93
<i>Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, Michael Young, Carnegie-Mellon Univeristy</i>	
An Extensible I/O System	114
<i>Jim Rees, Paul H. Levine, Nathaniel Mishkin, Paul J. Leach, Apollo Computer</i>	

Tools

Wednesday, June 11, 4:00 - 5:30

(Grand Salon)

PATHALIAS or The Care and Feeding of Relative Addresses	126
<i>Peter Honeyman, Princeton University, Steven M. Bellovin, AT&T Bell Laboratories</i>	
Pollster: A Document Annotation System for Distributed Environments	142
<i>Luis-Felipe Cabrera, IBM Almaden Research Center, Eric Mowat, University of California, Berkeley</i>	
When Network File Systems Aren't Enough: Automatic Software Distribution Revisited...	159
<i>Daniel Nachbar, Bell Communications Research</i>	

Networks 2

Thursday, June 12, 9:00 - 10:30

(Grand Ballroom)

Experiences Implementing BIND, a Distributed Name Server for the DARPA Internet	172
<i>James M. Bloom, CSRG, University of California, Berkeley, Kevin J. Dunlap, Digital Equipment Corporation</i>	
Network Performance and Management with 4.3BSD and IP/TCP	182
<i>Michael J. Karels, Marshall Kirk McKusick, CSRG, University of California, Berkeley</i>	
A Real-Time Electronic Conferencing System based on Distributed UNIX	189
<i>Tatsuo Suzuki, Hideo Taniguchi, Hisayasu Takada, NTT Electrical Communications Laboratories</i>	

Real Work 1

Thursday, June 12, 9:00 - 10:30

(Grand Salon)

How to Make Friends with Number-Crunchers	200
<i>Gregory Dudek, Michael Jenkin, Howard Marcus, University of Toronto</i>	
Kanji UNIX: Yunikkusu wa Nihongo o Hanasemasu (UNIX Speaks Japanese)	209
<i>Robert S. Jung, Joseph T. Kalash, Unisoft Systems</i>	
Tools for the Maintenance and Installation of a Large Software Distribution	223
<i>D. M. Tilbrook, P. R. H. Place, Imperial Software Technology</i>	

Distributed File Systems 1

Thursday, June 12, 11:00 - 12:30

(Grand Ballroom)

Vnodes: An Architecture for Multiple File System Types in Sun UNIX	238
<i>S. R. Kleiman, Sun Microsystems</i>	
Remote File Sharing Architectural Overview	248
<i>Andrew P. Rifkin, Michael P. Forbes, Richard L. Hamilton,</i>	
<i>Michael Sabrio, Suryakanta Shah, Kang Yueh, AT&T Information Systems</i>	
The Generic File System	260
<i>R. Rodriguez, M. Koehler, R. Hyde, Digital Equipment Corporation</i>	

Text Processing

Thursday, June 12, 11:00 - 12:30

(Grand Salon)

Modelling Text as a Hierarchical Object	270
<i>James Waldo, Apollo Computer Inc.</i>	
SMScript: An Interpreter for the POSTSCRIPT Language Under UNIX	284
<i>Bruno Borghi, Stephane Querel, Daniel de Rauglaudre, INRIA</i>	

Distributed File Systems 2

Thursday, June 12, 2:00 - 3:30

(Grand Ballroom)

The Network File System Implemented on 4.3BSD	294
<i>Ed Gould, Mt. Xinu</i>	
NFS Portability	299
<i>Mordecai B. Rosen, Michael J. Wilde, Lachman Associates Inc.</i>	
<i>Bill Fraser-Campbell, The Instruction Set, Ltd.</i>	
The Transparent Remote File System	306
<i>Ronald P. Hughes, Integrated Solutions, Inc.</i>	

Language Technology

Thursday, June 12, 2:00 - 3:30

(Grand Salon)

A Global Optimizer for Sun FORTRAN, C & PASCAL	318
<i>Vida Ghodssi, Steven S. Muchnick, Alex Wu, Sun Microsystems, Inc.</i>	
Four Generations of the Portable C Compiler	335
<i>David M. Kristol, AT&T Information Systems</i>	
The Notifier	344
<i>Steve Evans, Sun Microsystems, Inc.</i>	

Distributed File Systems 3

Thursday, June 12, 4:00 - 5:30

(Grand Ballroom)

Error Recovery in a Stateful Remote Filesystem	355
<i>Alan Atlas, Perry Flinn, MASSCOMP</i>	
Distributed File Systems Panel	

Electronic Mail

Thursday, June 12, 4:00 - 5:30

(Grand Salon)

Mail Routing using Domain Names	366
<i>Craig Partridge, CSNET Coordination and Information Center</i>	
AT&T Mail	377
<i>Dale S. DeJager, AT&T Information Systems</i>	
A Mail File System for Eighth Edition UNIX	391
<i>David Hitz, Peter Honeyman, Princeton University</i>	

Operating Systems 2

Friday, June 13, 9:00 - 10:30

(Grand Ballroom)

Shared Libraries on UNIX System V	395
<i>James Q. Arnold, AT&T</i>	
Decreasing Realtime Process Dispatch Latency through Kernel Preemption	405
<i>David C. Lennert, Hewlett-Packard</i>	
MOS - Scaling Up UNIX	414
<i>Amnon Barak, On G. Paradise, Hebrew University of Jerusalem</i>	

USENET BOF

Friday, June 13, 9:00 - 10:30

(Grand Salon)

Windows

Friday, June 13, 11:00 - 12:30

(Grand Ballroom)

A Data-Flow Manager for an Interactive Programming Environment	419
<i>Paul E. Haeberli, Silicon Graphics, Inc.</i>	
UWM: A User Interface for X Windows	429
<i>Michael Gancarz, Digital Equipment Corporation</i>	
Programming with Windows on the Major Workstations	441
<i>Stephen Daniel, C. Durward Rogers, Microelectronics Center of North Carolina</i>	

Operating Systems 3

Friday, June 13, 11:00 - 12:30

(Grand Salon)

The Influence of Workload on Load Balancing Strategies	446
<i>Luis-Felipe Cabrera, IBM Almaden Research Center</i>	
A System V Compatible Implementation of 4.2BSD Job Control	459
<i>David C. Lennert, Hewlett-Packard</i>	
UNIX as a Virtual Machine Environment	475
<i>Robert E. Genter, BBN Laboratories Inc.</i>	

Real Work 2

Friday, June 13, 2:00 - 3:30

(Grand Ballroom)

Porting to a Network of Diskless Micros	486
<i>W. Appelbe, D. Coleman, A. Fratkin, J. Hutchison, W. Savitch,</i> <i>University of California, San Diego</i>	
A State-wide Distributed Computing System	499
<i>Tom Truscott, Bob Warren, Ken Moat, Research Triangle Institute</i>	
UNIX-based Distributed Printing in a Diverse Environment	514
<i>William E. Johnston, Dennis E. Hall, Lawrence Berkeley Laboratory</i>	

Wizard's Panel

Friday, June 13, 2:00 - 3:30

(Grand Salon)

AUTHOR INDEX

Accetta, Mike	93	Kleiman, S. R.	238
Appelbe, W.	486	Koehler, M.	260
Arnold, James Q.	395	Kristol, David M.	335
Atlas, Alan	355	Langston, Peter S.	13
Barak, Amnon	414	Leach, Paul J.	114
Baron, Robert	93	Lennert, David C.	405, 459
Bellovin, Steven M.	126	Levine, Paul H.	114
Bloom, James M.	172	Marcus, Howard	200
Bolosky, William	93	Martin, Bernard	46
Borghi, Bruno	284	McGrath, Gilbert J.	38
Cabrera, Luis-Felipe	142, 446	McKusick, Marshall Kirk	182
Clancy, Patrick	81	Mills, Philip M.	59
Coleman, D.	486	Mishkin, Nathaniel	114
Crossland, James	81	Moat, Ken	499
Daniel, Stephen	441	Mowat, Eric	142
DeJager, Dale S.	377	Muchnick, Steven S.	318
Dudek, Gregory	200	Nachbar, Daniel	159
Dunlap, Kevin J.	172	Olander, David J.	38
Evans, Steve	344	Paradise, On G.	414
Fenart, Jean Marc	46	Partridge, Craig	366
Fievet, Marc	46	Place, P. R. H.	223
Flinn, Perry	355	Querel, Stephane	284
Forbes, Michael P.	248	Rashid, Richard	93
Fraser-Campbell, Bill	299	Rauglaudre, Daniel de	284
Fratkin, A.	486	Rees, Jim	114
Gancarz, Michael	429	Remille, Annie	46
Genter, Robert E.	475	Rifkin, Andrew P.	248
Ghodssi, Vida	318	Rodriguez, R.	72, 260
Goldberg, David	28	Rogers, C. Durward	441
Golub, David	93	Rosen, Mordecai B.	299
Gould, Ed	294	Sabrio, Michael	248
Haeberli, Paul E.	419	Savitch, W.	486
Hall, Dennis E.	514	Shah, Suryakanta	248
Hamilton, Richard L.	248	Suzuki, Tatsuo	189
Hawley, Michael	1	Sznyter, Edward W.	81
Hitz, David	391	Takada, Hisayasu	189
Honeyman, Peter	126, 391	Taniguchi, Hideo	189
Hughes, Ronald P.	306	Taylor, Bradley	28
Huitema, Christian	46	Tevanian, Avadis	93
Hutchison, J.	486	Tilbrook, D. M.	223
Hyde, R.	260	Truscott, Tom	499
Israel, Robert K.	38	Vaysseix, Guy	46
Jenkin, Michael	200	Waldo, James	270
Johnston, William E.	514	Warren, Bob	499
Jung, Robert S.	209	Wilde, Michael J.	299
Kalash, Joseph T.	209	Wu, Alex	318
Karels, Michael J.	182	Young, Michael	93
		Yueh, Kang	248

PREFACE TO THE ATLANTA CONFERENCE

This conference marks the Eleventh Anniversary of Unix User's meetings. What began as informal gatherings among a handful of people with a common interest has grown into a semiannual event of major proportions. Gone are the days when individuals would raise their hands at the beginning of the meeting to indicate they had something of interest to share with the other attendees; our group is just too large for that to be practical any more. Informal presentations have been replaced by a formal review process complete with conference proceedings available to attendees when they register at the conference. Our meetings of less than 100 people have grown to fifteen to twenty times that size, requiring careful organization and coordination among many people, with plans often beginning more than two years before the actual conference date.

Clearly we have come a long way since those early days. If you're wondering where and when all those past conferences were held and who helped make them happen, here's a list:

<u>When</u>	<u>Where</u>	<u>Host</u>	<u>Size</u>
11-13 Jun 86	Atlanta, GA	Medical Systems Dev Corp (Dan Forsyth & Paul Manno)	?
15-17 Jan 86	Denver, CO	University of Colorado (Evi Nemeth)	1,157
10-14 Jun 85	Portland, OR	Tektronix (Steve Glaser)	1,730
23-25 Jan 85	Dallas, TX	(Charisse Castagnoli & Rob Kolstad)	1,200
13-15 Jun 84	Salt Lake City, UT	University of Utah (Randy Frank)	1,500
16-20 Jan 84	Washington, DC	* (Reidar Bornholdt)	8,000
13-15 Jul 83	Toronto, ON	HCR (Mike Tilson)	1,500
26-28 Jan 83	San Diego, CA	University of California at San Diego * (Tom Uter)	1,850
6-9 Jul 82	Boston, MA	BBN (Alan Nemeth)	1,300
27-29 Jan 82	Santa Monica, CA	ISC (Mike O'Brien)	1,000
24-26 Jun 81	Austin, TX	University of Texas (Wally Wedel)	500
21-23 Jan 81	San Francisco, CA	University of California at San Francisco (Tom Ferrin)	1,000
17-20 Jun 80	Newark, DE	University of Delaware (Dan Grim)	
29 Jan-1 Feb 80	Boulder, CO	National Center for Atmospheric Research (John Donnelly)	450
30 Nov 79	Menlo Park, CA	SRI (John Bass)	
20-23 Jun 79	Toronto, ON	University of Toronto (Ron Baecker)	400
25-27 Jan 79	Santa Monica, CA	ISC (Steve Holmgren)	350
2 Oct 78	Menlo Park, CA	SRI (John Bass)	75
24-28 May 78	New York, NY	Columbia University (Lou Katz)	~350
12-13 Sep 77	Menlo Park, CA	SRI (Oliver Whitby & John Bass)	~100
19-21 May 77	Urbana, IL	University of Illinois (Steve Holmgren)	~250
1-3 Oct 76	Cambridge, MA	Harvard University (Lewis Law)	~120
1-2 Apr 76	Cambridge, MA	Harvard University (Lewis Law)	~60
27-28 Feb 76	Berkeley, CA	University of California at Berkeley (Bob Fabry)	
31 Oct 75	Monterey, CA	Naval Postgraduate School (Belton Allen)	
27 Oct 75	New York, NY	City University of New York (Mel Ferentz)	
18 Jun 75	New York, NY	City University of New York (Mel Ferentz)	40

*Indicates joint conference with /usr/group.

(This preface and table were shamelessly stolen from the USENIX Conference Proceedings, Summer 1985.)

MIDI Music Software for UNIX[†]

Michael Hawley

The Droid Works
P.O. Box CS 8180
San Rafael, CA 94912

ABSTRACT

This paper describes MIDI music software for 4.2BSD UNIX. We use a Sun workstation to drive a real-time processor controlling MIDI synthesizers. The musical examples may seem somewhat naive — viewing and editing keystroke graphs, playing Bach upside down, transforming text into music, an old-time musical dice game — but they inkle at the potential of using music to inspire creativity and happiness.

This work originated with Gareth Loy at UCSD. The multibus interface has been reworked and the library and application-level software has grown substantially. The music software is in the public domain.

1. Introduction

Computer music has hit the streets. It is now possible to buy relatively inexpensive (\$1000 or less) synthesizers of surprising and beautiful timbral complexity that also interface to computers. The interface is MIDI — the musical instrument digital interface, both a hardware standard and software protocol, something like RS232 and ASCII for synthesizers. For example, a PC and a good digital piano (about \$3000 total) would constitute a system that is in many ways more interesting than the vacuum-powered Aeolian or Ampico reproducing player pianos of the 1930's. Of course, a “good” reproducing player piano requires a high-quality grand piano as well as a player mechanism that is about as complicated and reliable as an oil refinery.[‡] On the other hand, a reasonable chamber orchestra of synthesizers can be assembled for less than the cost of a grand piano and offer acoustical and interactive possibilities that go far beyond traditional mechanical music.

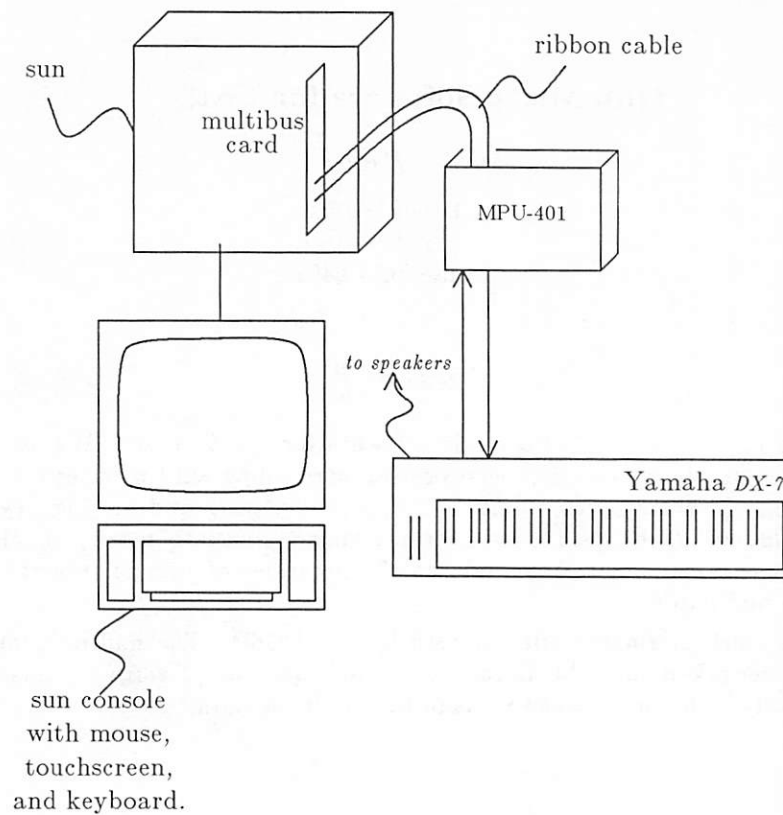
Following is a description of our UNIX/MIDI setup, discussing hardware, libraries, and some simple applications.

2. Hardware

We put a multibus board into a Sun-2 to connect it to the Roland MIDI processing unit (MPU-401), known as `/dev/mpun` to a UNIX driver. This device connects (through MIDI cables) to MIDI synthesizers (currently a Yamaha *DX-7*; we have also used drum machines and other synthesizers):

[†] UNIX is a trademark of AT&T Bell Laboratories.

[‡] Not quite true, since the wheezing bellows and air motors have been succeeded by electrical devices. The best player piano in the world is a Bösendorfer concert grand (about \$60,000) instrumented with a microprocessor controller and a system of solenoids and optical sensors (about \$20,000) designed by Wayne Stahnke.



The Roland MPU-401 is a small, cheap (about \$200) MIDI processor. It was designed for use with personal computers and is capable of controlling 16 MIDI channels (i.e., instruments). It typically manages real-time problems like playing and recording. For instance, scores consist of time-tagged lists of events — key on, key off, parameter changes, etc. — and to play a score, an application simply writes data to the device. The MPU-401 buffers and plays score data, interrupting the host when more data is needed.

The multibus board design (and a PAL program for it) came from Gareth Loy *et al.* at UCSD.[†] For the benefit of UNIX hobbyists who want a quick start, here is the PAL program (in *palasm*) and multibus board plan:

```

pal1618
tompal2
MPU-401 to multibus-1 interface card
cme/carl san diego, ca    12-17-84
/e /iorc /iowc q /adr3 /adr4 /adr5 /adr6 nc gnd
nc /g /t /r /xack /en /mr nc nc vcc

g = /adr3*/adr4*/adr5*/adr6
t = e*iorc
r = e*iowc
en = t+r
if (en) xack = q
mr = /en

function table

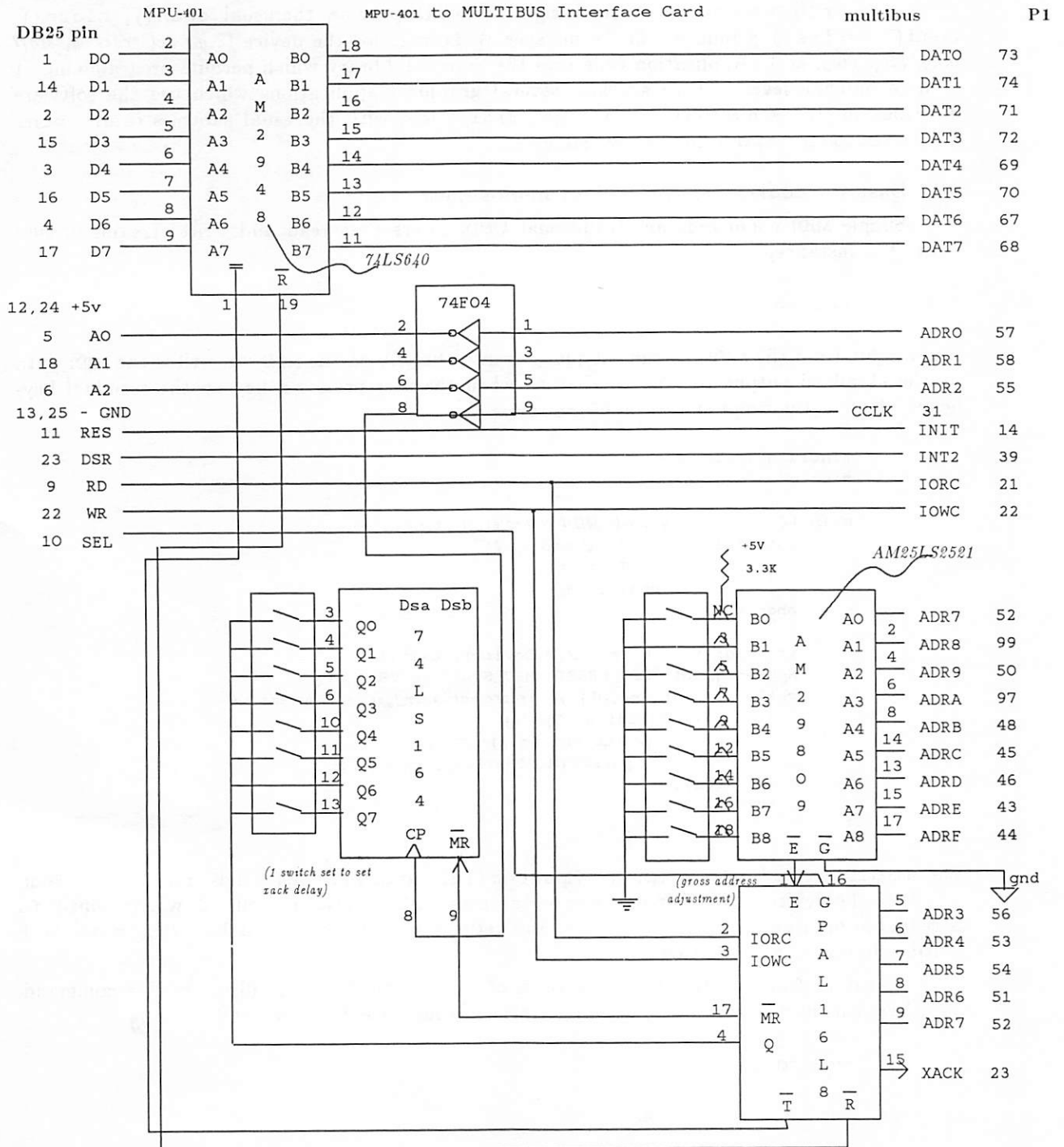
```

[†] We have since switched to a simpler off-the-shelf multibus i/o board which needs only a little wire wrapping. We understand that Gareth has redesigned the multibus board for VME bus.

/e /iorc /iowc q /adr3 /adr4 /adr5 /adr6 /g /t /r /xack /en /mr

x	x	x	x	h	h	h	h	1	x	x	x	x
x	x	x	x	h	l	h	l	h	x	x	x	x
l	l	h	l	x	x	x	x	x	l	h	h	l
l	l	h	h	x	x	x	x	x	l	h	l	l
l	h	l	l	x	x	x	x	x	h	l	h	l
l	h	l	h	x	x	x	x	x	h	l	l	l
h	l	h	l	x	x	x	x	x	h	h	z	h
h	l	h	h	x	x	x	x	x	h	h	z	h
h	h	l	l	x	x	x	x	x	h	h	z	h
h	h	l	h	x	x	x	x	x	h	h	z	h

And the board:



Notes on the board.

We used some 22uf/25v axial capacitors and some filtering/bypass capacitors.

Peter Langston (at Bell Communications Research) observes that since address information is preserved in the PAL, a trivial change in the PAL program and a few more connectors permits control of 4 MPU-401's simultaneously. (Of course, using this would require rewriting the driver).

In lieu of store-bought MIDI cables, we use 5-pin DIN connectors with modular phone jack plugs. Telephone wire is fragile on stage but convenient in the lab.

3. Software

A driver (first written by Rusty Wright at UCSD) provides the usual `open()`, `close()`, `read()`, `write()` primitives, and some special `ioctl`s for the device (*e.g.*, *set track #*, *start play*, *stop play*, etc). Application code uses the `-lmidi` library which permits programming at a more humane level. There are also several graphical applications which use the software described in [1]. A directory called `/usr/midi` exists, with the usual subtrees (*man*, *bin*, *include*, *lib*, *src/bin*, *src/lib*, etc).

3.1. Basics: *record*, *play*, *da*, *transpose*, etc.

Simple MIDI commands are traditional UNIX filters that read and write streams of MIDI data. For instance,

```
record > foo
```

reads input from the MIDI instrument plugged into MIDI *IN* on the MPU-401, writes the MIDI data on the standard output to *foo*, and stops when the user presses a key on the terminal keyboard. Here is the simplest version of *record*:

```
#include <stdio.h>
#include <midi.h>

main(){ /* a very simple MIDI record program */
    int midi = open(MidiDevice, 2),
        out = fileno(stdout),
        kbd = fileno(stdin);
    char c;

    if (midi < 0) perror(MidiDevice), exit(1);
    MpuSend(midi, MPU_RESET, MPU_START_RECORD, 0);
    while (!iwait(kbd,0)) /* record until user types on keyboard */
        if (read(midi,&c,1)==1)
            if (write(out,&c,1) != 1)
                perror(MidiDevice), exit(1);
    close(midi);
    exit(0);
}
```

The `MidiDevice` is `/dev/mpu0`. `MpuSend(fd, args..., 0)` sends initialization commands to the device; `iwait(fd, nseconds)` uses `select()`(2) to poll `fd`, waiting until `fd` is readable or `nseconds` have elapsed, and returns true if `fd` is readable. (These are both routines in the `-lmidi` library).

MIDI data from the MPU-401 is a stream of time-tagged bytes. A disassembling command, *da*, filters this into something human-digestible. For instance, the command

```
record | da
```

produces verbose ASCII output like

32	90	3c	38 ;	0.417	0	kon	[60]=56	C4 key on
17		3c	0 ;	0.608	1	koff	[60]=0	C4 key off
52		3e	38 ;	1.292	2	kon	[62]=56	D4 key on
18		3e	0 ;	1.492	3	koff	[62]=0	D4 key off
57	40	49 ;		2.217	4	kon	[64]=73	E4 key on
10	40	0 ;		2.350	5	koff	[64]=0	E4 key off
6f	3c	40 ;		3.275	6	kon	[60]=64	C4 key on
a	3c	0 ;		3.358	7	koff	[60]=0	C4 key off
4d	f9			4.000	8	tcwme	[0]	timing clock with measure end
	f8			8.000	10	tcip		timing clock in play

The MIDI data are the *hex* bytes to the left of the semicolon (at most four bytes per command); to the right is its disassembly, with a running count of time in seconds for each event. The first byte is a time tag, the second is an optional MIDI command (the previous command is *sticky*), and the third and fourth bytes are command-specific data (e.g., <pitch><velocity>, in the case of key on/off events, where *velocity*=0 indicates *key off*).

The `play` command writes files of MIDI data to the MPU-401, which plays them. `Play` takes several command line flags (e.g., to set tempo, MIDI channel number, play several streams in parallel, overdub, etc). To play MIDI scores, some data is pre-written to the device, the device is then placed in *play* mode (playing begins), and the rest of the data is written to the device. The command

```
play foo
```

plays the file `foo`. The dis-assembled data above can be reassembled and piped to `play`:

```
da foo | sed 's/;.*//' | atox | play
```

A re-assembling command, `ra`, regenerates the MIDI time tag (the leftmost byte) from the computed floating-point time in the third column; thus it is possible (but grody to hand-edit and sort `da` listings when necessary).

The `play` command has options to record (`-r`), and to play files simultaneously (`-s`):

```
play -r -s soprano alto bass > tenor
```

plays three voice parts and records any data from the synthesizer keyboard onto a new file called `tenor`. There are similar filters to transpose, stretch, merge, retrograde, and otherwise massage MIDI data. For instance:

```
transpose +7 file | stretch -f .5 | play
```

transposes `file` up a fifth (i.e., 8 semitones) and plays it at twice normal speed (compressing event timings by a factor of .5). Transpose is a good example of a simple note filter:

```
TransposeMpuCmd(m, shift)
    MpuCmd *m;
/* Transpose 'm' by 'shift' half steps
   (e.g., if 'shift' is '-7', 'm' is transposed down a fifth.
*/
{
    if (IsNote(m))
        MpuPitch(m) += (unsigned char) shift;
}

MidiTranspose(in,out,shift)
    FILE *in, *out;
/* Copy the midi data from 'in' and transpose it by 'shift' half steps, writing the result in 'out'.
   Example: 'MidiTranspose(in,out,-7)' transposes down a fifth.
```



```

*/
{
    MpuCmd m;
    while (GetMpuCmd(in,&m))
        TransposeMpuCmd(&m,shift), PutMpuCmd(out,&m);
}

```

In addition, the `play` command (and other commands) can accept piped streams as well as filenames, *e.g.*:

```
play -s "|cmd1..." "|cmd2..." ...
```

where the notation `"|cmd1..."` means "read from the command `cmd1...`" This allows both files and live processes to provide streams of input. The notation is implemented by a simple pair of routines, `sopen()` and `sclose()`, which combine `fopen/fclose` and `popen/pclose`. `sopen(name,mode)` opens `name` as a pipe, using `popen()` if `name` begins with a bar `"|"`; otherwise `fopen()` is used. Infix commands can execute on remote systems, so processing can be done in parallel. This makes it possible to write commands like:

```
play -s v1 "|transpose +7 v1" v2 "|transpose -1o v2"
```

which plays four parts: `v1` and its transposition up a fifth, and `v2` with its transposition down an octave. This facility has been useful in many other contexts.

3.2. Generating MIDI Data: Scales and Random Notes

The `-lmidi` library provides an `MpuCmd` data structure and a few primitives for dealing with notes: `MpuNote(pitch,velocity,delay)` creates a *note on* or *note off* command; `SetMpuNote(m,pitch,velocity,offset)` sets a *note's* data; and `PutMpuCmd(f, m)` writes an `MpuCmd` to FILE `*f`. The following program, `scale.c`, writes out a scale of notes with a gradual crescendo:

```

#include <stdio.h>
#include <midi.h>
#define note(a,b,c) PutMpuCmd(stdout, SetMpuNote(m,a,b,c))
MpuCmd *MpuNote(), *SetMpuNote();

#define dx7_MIN 36 /* range of notes on dx7 keyboard */
#define dx7_MAX 97

main(){
    MpuCmd *m = MpuNote(0,0,0); /* to hold a MIDI note */
    int pitch,
        velocity = 10,
        duration = 25; /* in MIDI ticks */

    for (pitch=dx7_MIN; pitch<dx7_MAX; pitch++) {
        note(pitch, velocity, 0); /* note ON */
        note(pitch, 0, duration); /* note OFF after duration ticks */
        velocity = (velocity<99)? velocity+1 : velocity;
    }
}

```

And to play it:

```
scale | play
```

MIDI commands can also be issued directly by the host without using time-tagged note lists (*i.e.*, bypassing the MPU-401's sense of real time). For instance, here is a program that plays random

notes, using NoteOn(fd, track, pitch, velocity) and NoteOff():

```
#include <midi.h>

float Tempo = 100., atof();
int midi;
#define rnd(a,b)      (random()%(b-a)+a)    /* return # in range a...b */

Error(s){ MidiError("%s: ", av0), perror(s), exit(1); }

RandomNote()
/* Play a random note with random velocity (amplitude) for a random time:
   turn the note on, pause a random bit, then turn it off.
*/
{
    int p = rnd(dx7_MIN, dx7_MAX);    /* a random dx7 note */
    NoteOn(midi, 0, p, rnd(20,99));
    fsleep((float)(rnd(10,100))/Tempo);
    NoteOff(midi, 0, p);
}

main(ac, av) char *av[]; {
    int i, n = 100;

    srand(42);    /* 42 is the most random number. */
    for_each_argument {
        Case 'n': n = atoi(argument);    /* number of notes to play */
        Case 'r': srand(atoi(argument)); /* use argument as seed */
        Case 't': Tempo = atof(argument);
        Default : MidiExit(1,"use: %s [-n #notes] [-r seed] [-t tempo]0,av0);
    }

    if ((midi = open(MidiDevice, 2)) == -1) Error(MidiDevice);
    MpuSend(MPU_RESET, MPU_MIDI_THRU_OFF, 0);
    printf("Instant Boulez...0);
    for(i=0; i<n; i++) RandomNote();
    printf("Applause...0);
    close(midi);
    exit(0);
}
```

To do this, the MPU-401 accepts a *want_to_send_data* command which allows subsequent MIDI commands to be written directly to the device (*i.e.*, the host worries about timing). In practice, this usually requires reserving one of the MPU-401's 8 play tracks for this purpose.

3.3. muzak: An ASCII → MIDI Filter

The muzak program filters ASCII text to music by playing ASCII values as notes, which turns out to be fairly amusing. The basic filter is simple:

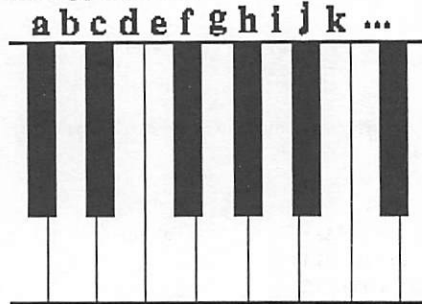
```
int MIDI; /* file descriptor of Roland device */

note(n){ /* play pitch n */
    int velocity = 100;
    NoteOn( MIDI,n,velocity );
    nap(Tempo); /* sleep for Tempo 60ths of a second */
    NoteOff( MIDI,n );
}

main(){
    char c;
    MIDI = open("/dev/midi",2);
    if ( MIDI < 0 ) perror("no MIDI - bleagh!"), exit(1);
```

}

In other words, there is a mapping like:



The actual program is fancier, but not much. It has options to do things like play punctuation characters longer than letters, play louder and faster inside parenthesized groups (nested loops in C or LISP code sound frantic), etc. The following makes a cute, even beautiful, tune:

```
hip... hip... hippety hip hap... hap...

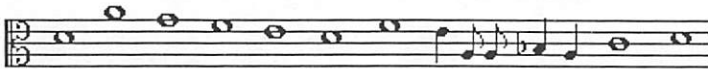
happy birthday to you
happy birthday to you
happy birthday dearrrr aannnnndddddyyy Y Y Y Y Y Y . . .
happy birthday to... [you] {!!!}
```



Playing kernels and other binaries tends to be tedious and spastic, but alphabetized listings, hyper-punctuated dictionary definitions, and numeric files (like octal dumps) are driving and exciting. With text, letter frequencies give tonal colors to the sound — happy birthday is in **c#** minor with a bitonal hint of **D** major, and the similarity between the melody happy (**G#C#E-E-C#**) and ...hday (**G#E-C#C#**) is musically interesting (and serendipitous).

3.4. dice: a Simple Music Language

The `dice` command uses a `yacc`-based parser to translate a very simple textual music language to MIDI. For instance, the following counterpoint exercise:



is written as:

voice 0 d a+ g f e d f e.4 a-.8 a bb.4 a c.4 d

a+ means a in the next higher octave (the nearest note is used by default, which would have been a a fourth below d; a+ forces a a fifth above); time values are written .nn, where nn is 1, 2, 4, 8, etc, for whole notes, half notes, quarter notes, eighth notes, etc; accidentals are

written as #, b, x, bb, n for sharp, flat, double sharp, double flat, natural. The language also has a construct whereby lines beginning with "#|"

```
#| /lib/cpp | sed -f abbreviations
```

cause remaining input to be filtered through the given UNIX command, to allow arbitrary kinds of macro processing.

The language is too simple to be of really general use, but we have typed in the music for Mozart's 1798 *Muzikalisches Würfelspiel* (musical dice game), one of the earliest known pieces of computer music. Here are the instructions from the original edition:

INSTRUCTION.

To compose, without the least knowledge of Music, Country-dances, by throwing a certain Number with two Dice.

- 1.) The Letters A--H, placed at the head of the 8 Columns of the Number-Tables show the 8 times of each part of the Country-Dance, viz. A, the first, B, the second, C, the third, &c: and the Numbers in the Column under the Letters show the Number of the time in the Notes.
- 2.) The Numbers from 2, to 12, show the sum of the Number than can be thrown.
- 3.) For instance, in throwing for the first time of the first part of the Dance, with two Dice, the Number 6, one looks next to that Number in the Column A, for the 10th time in the Notes. This time is written down, and makes the beginning of the Dance. — For the second time, for instance, the Number 8, being thrown, turn to the same table Column B, and the Number 81 shall be found. This time is put next to the first, & one continues, in this manner till the eight times shall be thrown, when likewise the first part of the Dance shall be finish'd. — The sign of repetition is further placed & the second part begun.

	A	B	C	D	E	F	G	H
2	70	14	164	122	25	153	18	167
5	10	64	100	12	149	30	161	11
4	33	1	160	163	77	156	168	172
5	36	114	8	35	111	39	197	44
6	105	150	57	75	117	52	192	130
7	165	152	112	15	147	27	73	102
8	7	51	131	37	21	125	49	115
9	142	106	40	69	43	140	25	89
10	99	68	86	139	120	92	143	83
11	55	45	90	158	82	123	78	58
12	145	27	6	151	55	67	57	16

Centerdances Angloises.



The "score" looks like this:

```
#| uncomment | /lib/cpp | sed -f abbrevs
(in abbrevs, = means .16, _ means .8, and ~ means .4)
```

2/4, 3 voices

```
voice 0
g5_ g e c
d f b- d
g b= g d~
...
```

A random bit of Mozart is then played by executing a command like

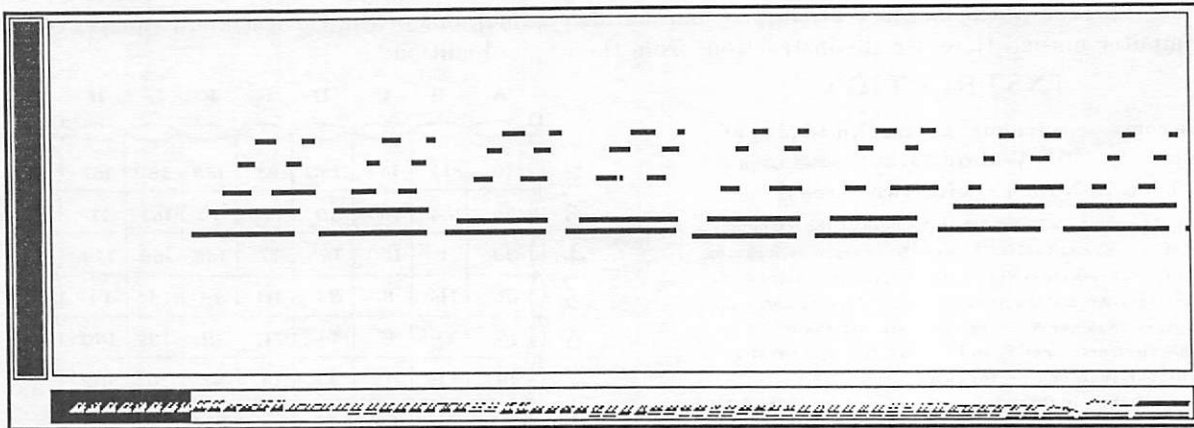
```
roll | dice countrydance.score > wolf
play wolf
```


3.5 Upside-Down Bach

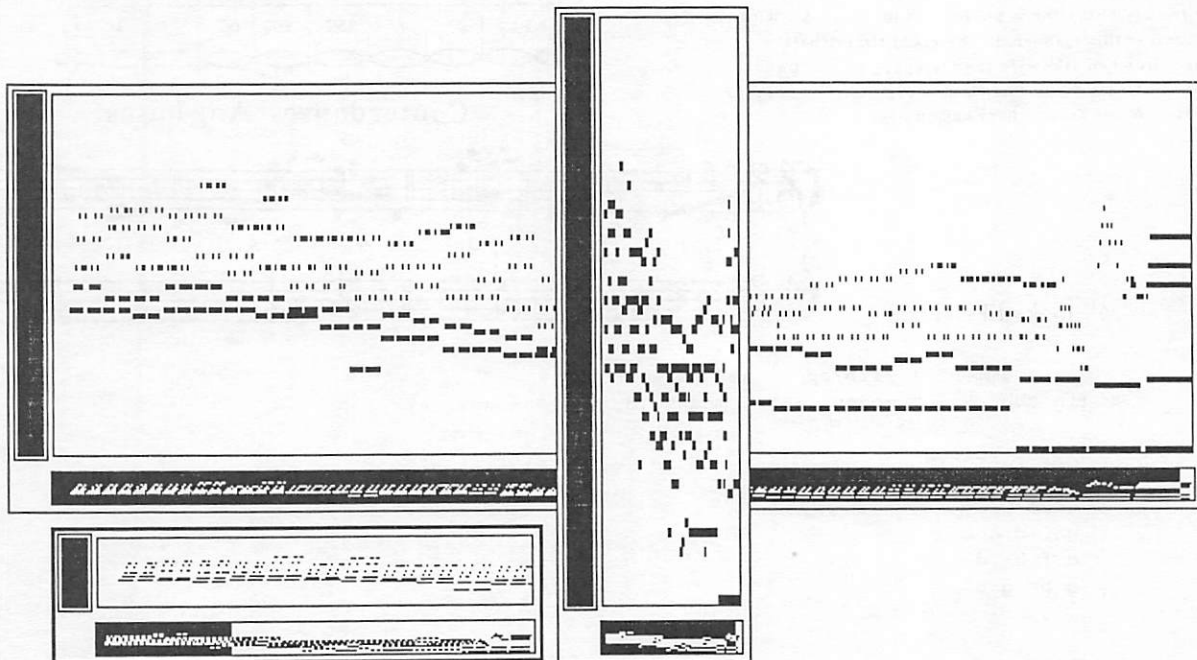
The first prelude from Book I of Bach's *Well Tempered Clavier*



played into a computer looks like this:



This editor displays score data in piano roll format; the scrollbar gives a bird's-eye view of the score, showing the whole piece and the part in view. We can zoom in or zoom out, or by reshaping the frames, look at the music in a variety of new ways:



We invert the piece by mapping pitches, p , in the range $a \dots b$ linearly to an output range $A \dots B$:

$$((B-A) * (p-a)) / (b-a) + A$$

This simple filter does that:

```

#include <stdio.h>
#include <midi.h>
#define Alloc(x) (x *)calloc(1,sizeof(x))
#define md(a,b,c) ((long)(a)*(long)(b))/(long)(c)
#define transform(v, a,b, A,B) md(B-A, v-a, b-a) + A

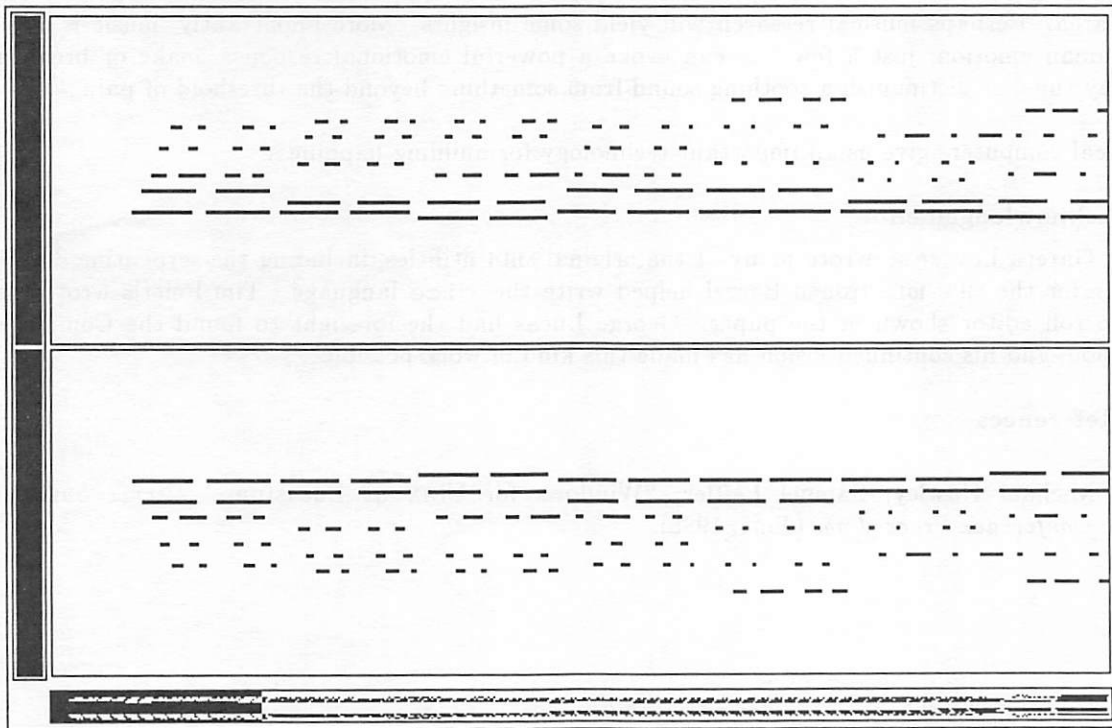
between(a,x,b){ return a <= x && x <= b; }

main(){
    MpuCmd *m = Alloc(MpuCmd);
    char p,
        a=dx7_MIN, b=dx7_MAX, /* input range */
        A=dx7_MAX, B=dx7_MIN; /* output range */

    while (GetMpuCmd(stdin,m)) {
        if (IsNote(m) && between(a,p=MpuPitch(m),b))
            MpuPitch(m) = transform(p, a,b, A,B);
        PutMpuCmd(stdout,m);
    }
}

```

Inverting over the entire keyboard range is like playing on a keyboard with high notes at the bottom and low notes at the top. The rightside-up and upside-down versions together look like this:



which is not visually surprising, but this particular piece has an eerie beauty when played upside down. On the other hand, Art Tatum arrangements sound horrible this way, because keyboard and melodic idioms don't invert as well as harmonies — there is more to this class of composition than simply running random material through an informational meat grinder. Contrapuntal inversion has long been a popular, if academic, compositional device, but full *harmonic inversion* hasn't happened before: without a computer, it's just too tedious to rewrite the notes. Inverting harmonies maps major into minor, and vice-versa, and classically grammatical harmonic progressions are transformed into modal ones. For example, a garden-variety conclusive I-IV-V-I dominant→tonic major cadence (e.g., C-F-G-C) when turned upside down becomes modal and minor i-v-iv-i (f-c-b^b-f). The result is a music that sounds at once both old (because of

the modalisms) and very new at the same time, with a weird familiarity.

4. Conclusions

Worthwhile musical appliances are now cheap, sound good, and are computer controllable. A UNIX/MIDI environment is relatively easy to construct, and all the software described in this paper is in the public domain. The setup described here, or variants of it, exists at perhaps a dozen sites around the world. Source code, MIDI tricks, and synthesizer voices are regularly traded by computer mail.

A great psychologist has observed that "technology has evolved with precious little direct attention to happiness. It has usually been intended to make people safe, fast, rich, and good at killing each other. The new [information] technology offers us the tools to change the way people spend their workdays, exercise their skills, intelligence, and creativity, and conduct their interpersonal relations." New technology invites new discovery, and new discovery requires new technology. That is reason enough why every computer research group should have easy access to a room full of synthesizers. Persuading computers to make music pushes many interesting computational problems — real time programming, parallel programming, signal processing, control theory, user interfaces (*e.g.*, building a machine to accompany soloists), the mapping between graphical languages (common music notation) and performances, etc. Music also directly touches a fundamental and very badly understood factor in human learning and memory: temporal spacing. It is known that the rhythm and frequency of presentation of a stimulus affect our retention in profound ways, but little quantitative work has been done in this area. Perhaps musical research will yield some insights. More importantly, music is close to human emotion: just a few bits can evoke a powerful emotional response, make or break a catchy tune, or distinguish a soothing sound from something beyond the threshold of pain.

Musical computers give us an important technology for building happiness.

5. Acknowledgements

Gareth Loy *et al.* wrote many of the original MIDI utilities, including the serpentine device driver for the MPU-401. Ronen Barzel helped write the `dice` language. Tim Peierls wrote the piano roll editor shown in the paper. George Lucas had the foresight to found the Computer Division, and his continued vision has made this kind of work possible.

6. References

- [1] Michael Hawley, Samuel Leffler, "Windows for UNIX at Lucasfilm," *Usenix Summer Conference Proceedings* (June, 1985).

(201) 644-2332
or
Eddie & Eddie on the Wire
An Experiment in Music Generation

Peter S. Langston

Bell Communications Research
Morristown, New Jersey

ABSTRACT

At Bell Communications Research a set of programs running on loosely coupled Unix systems equipped with unusual peripherals forms a setting in which ideas about music may be "aired". This paper describes the hardware and software components of a short automated music concert that is available through the public switched telephone network. Three methods of algorithmic music generation are described.



Introduction

Ten years ago, in order to experiment with computer-generated music, a researcher would have required a large computer and a great deal of special purpose equipment or would have had to settle for orchestrating the serendipitous squeaks and squawks of other equipment.⁰ In the last few years, advances in signal processing techniques and large-scale integration, combined with the proliferation of consumer music products have brought the cost of computer-controlled music hardware down to that of conventional musical instruments. Sound processing hardware is now reaching a level of availability comparable to that reached by text processing hardware ten to fifteen years ago. In the next ten years it would not be unreasonable to expect intense activity in the area of sound manipulation software, with a revolutionary impact on industries that depend on sound.

Sound can be used in many ways; the single most important use is as a communications medium. Spoken language is the most obvious of the techniques for communication via sound, and it has been the subject of intense research for many years. Another use of sound is to create an experience that "communicates" on a non-linguistic level.

"Music" incorporates both communication and experience but the line separating them becomes somewhat indistinct. What is it we enjoy in a piece of music? There are convincing arguments to the effect that music has both a vocabulary and a grammar, with different types of music having different vocabularies and grammars. Thus, the enjoyment of music becomes a learned skill, much like reading French or Latin, and new types of music appear to be gibberish until they are learned, e.g. "Gamelan music is Greek to me." There are many interesting questions to be answered:

⁰ In 1965 I even wrote a compiler to turn music notated on punched cards into a program that, when run, would produce controlled radio interference and thereby "play music" on a nearby transistor radio.

What is the language of music? Does it have a grammar? What differentiates a "jumble of notes" from "a piece of music"? What are the semantics of music? Can you say "The countryside is very peaceful" or even "I have lost my American Express card" in music?

Before we can hope to answer any of these questions we need to have a lot more data. The present project seeks to provide some data for questions about differentiating "music" from "a jumble of notes" through subjective evaluation of the output of programs that assemble notes by relatively simple rules. Hopefully we will be able to draw some conclusions from these data. For instance: a) If the program outputs are deemed "musical" then the rules used in the program are *sufficient*, b) If the outputs of programs with non-overlapping sets of rules are deemed "musical", then neither set of rules is *necessary*.

The approaches used here for music generation deal only with syntactic (i.e. form) considerations; the programs have no methods for handling semantics. The semantics of music are assumed to involve an immense wealth of cultural and historical information (a.k.a. "knowledge") that does not yet exist in any machine readable form. Understanding the semantics of music is no simpler than understanding the semantics of natural languages; for that matter, it would be easy to argue that music is a natural language, albeit often a non-verbal one.

Concurrent with the project in music generation at Bell Communications Research is a project exploring the benefits of interconnecting computers and telephone equipment [REDMAN85] [REDMAN86]. As a demonstration of both projects, an E&M trunk (a direct audio connection) on our experimental telephone switch was allocated to provide telephone access to the music hardware. Some programs were then written to present, in an entertaining form, examples of the music generated.

(201) 644-2332

A block of 100 telephone numbers (644-2300 through 644-2399) have been allocated to an experimental telephone switch in our lab which has been dubbed "BerBell" (the switch itself is manufactured by Redcom, Inc., which has no direct relationship to B. E. Redman, logname "ber", the principal investigator on the BerBell project). Aside from providing enhanced telephone service for approximately 40 "people" lines, (32 in-house extensions, two remote lines to residences, and several numbers to which participants can forward their home phones to use BerBell's call management features), a number of experimental services are provided.

Among these services are:

644-2300	BerBell robot operator
644-2311	Touch-Tone Directory Assistance [MORGAN73]
644-2312	games (name the beast, guess your age)
644-2312	miscellaneous information (quote of the day, today's events in history)
644-2331	recorded music of the day
644-2332	Eddie & Eddie on the wire (the topic of this paper)
644-2335	news summary (derived from the Associated Press news wire)
644-2337	weather report (derived from the National Weather Service wire)

and others.

Figure I shows the various hardware modules that form the system and their interconnections. Lines terminated with square boxes are EIA RS-232 serial connections (bidirectional). Lines terminated with simple arrowheads are audio connections (directional). Lines terminated with pentagonal arrowheads are Musical Instrument Digital Interface (MIDI¹) connections (directional). The remaining connections are either Ethernet lines (terminated with rectangles) or telephone lines (with no special termination symbol). The E&M trunk mentioned earlier connects the equalizer in

¹ MIDI is a standard representation of synthesizer "events" (e.g. note-on, note-off, volume change, etc.) as a serial data stream [MIDI85].

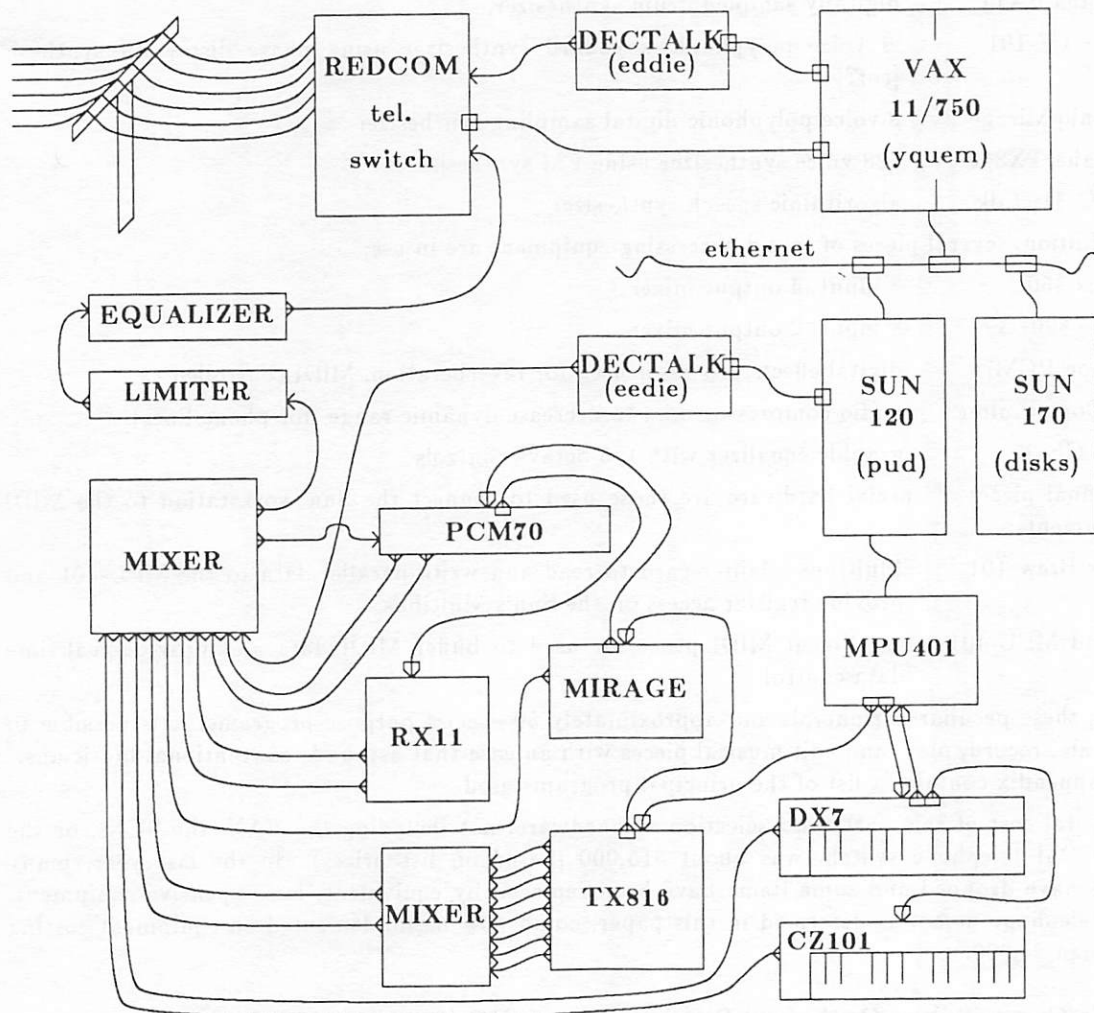


Figure I - Simplified interconnection schematic

the upper left corner of the figure with the Redcom telephone switch.

When a call comes in to 644-2332 a process called "demo2332" is spawned on the VAX 11/750 ("yquem" in figure I). This process first connects the Dectalk speech synthesizer ("Eddie" in figure I) to the caller's incoming line and then proceeds to introduce the demo, ask the caller to enter his/her telephone number (not for billing, but for some idea of the demographics of the participants), and to choose a long, medium, or short demo. Concurrently, "demo2332" spawns another process that probes the Sun workstation ("pud" in figure I) to determine whether or not it is running (crashes have not been uncommon). If the probe finds pud "up", a remote process is started on pud to carry out the demo and the caller's incoming line is disconnected from Eddie and is reconnected to the equalizer in the lab where the sound generation hardware is installed. The new process running on pud plays a short fanfare to make sure that the sound generation equipment is functional (it sometimes fails for obscure reasons) and then introduces and plays each piece. The introductions are spoken by another Dectalk voice synthesizer ("Eddie" in figure I). Note that the Dectalk does a credible job of transforming ASCII text into speech, but that sometimes creative misspelling is required in order to resolve ambiguous or irregular cases (e.g. "Eddie"). The music is played on sound synthesizers communicating via MIDI. At the moment the sound synthesizers in use are:

Yamaha DX7 16 voice polyphonic keyboard synthesizer using FM synthesis [CHOWN-ING73]

Yamaha RX11	digitally sampled drum synthesizer,
Casio CZ-101	8 voice polyphonic keyboard synthesizer using phase distortion synthesis [ref?]
Ensoniq Mirage	8 voice polyphonic digital sampling synthesizer
Yamaha TX816	128 voice synthesizer using FM synthesis
D.E.C. Dectalk	algorithmic speech synthesizer

In addition, several pieces of sound processing equipment are in use:

Fostex 450	8 input, 4 output mixer
Tapco 8201B	8 input, 2 output mixer
Lexicon PCM70	digital effects processor used for reverberation, MIDI controlled
S-S Compliter	audio compressor used to decrease dynamic range (for phone lines)
Rane GE-27	graphic equalizer with 1/3 octave controls

The final pieces of special hardware are those used to connect the Sun workstation to the MIDI instruments:

Home-Brew 101	Multibus adapter card to read and write parallel data to the MPU-401 and provide register access on the Sun's Multibus
Roland MPU-401	intelligent MIDI processor used to buffer MIDI data and provide real-time data control

Using these peculiar peripherals and approximately 50 special purpose programs, it is possible to generate, record, play, and edit musical pieces with an ease that astounds conventional musicians.² The appendix contains a list of the principal programs used.

The total cost of this particular selection of hardware, not including the VAX, the SUNs, or the REDCOM telephone switch, was about \$15,000 (based on list prices). In the last year, many prices have dropped and some items have been replaced by equivalent, less expensive equipment. The telephone demo, as described in this paper, could now be implemented on equipment costing less than \$4,000.

Music Generated by Optimized Randomness — Riffology

Schemes for harnessing random events to compose music predate computers by many years. Devised in the eighteenth century, Wolfgang Amadeus Mozart's "Musical Dice Game" gave an algorithm for writing music by rolling dice. Since that time, researchers at the University of Illinois, Harvard University, Bell Telephone Laboratories, and numerous other places have replaced dice with computers and turned disciplines such as signal processing, combinatorics, and probability theory toward the task of composing music [HILLER70].

The idea for the first music generation technique used in the telephone demo did not come from these illustrious predecessors, however. It came from experiences as lead guitarist in several bands that performed improvisational music. One of the popular criticisms that could be levelled at another guitarist was that he or she "just strings a lot of riffs together and plays them real fast". Of course, you were supposed to be pouring out your soul and a little bit of divinely-inspired ecstasy instead.³ The principal objection to playing an endless succession of "riffs" is that it doesn't involve a great deal of thought, just good technique. That is to say, the syntax is more important than the semantic content. For this reason, an algorithmic implementation of riffology need not be hampered by its inability to manipulate semantics.

The theme music for the video game "*ballblazer*" (tm, Lucasfilm Ltd.), called "Song of the Grid" (tm), is generated by just such an approach [LEVINE84] [LANGSTON85]. The program runs in little memory on a small 8-bit processor (a 6502) connected to a sound chip that can make 4

² These are *tools* for musicians, however, not *replacements* for them.

³ I was never accused of this form of "riffology" myself; I always stuck to straight soul-pouring.



Figure II - Riffology from "Song of the Grid"

independent sounds at once using square waves and pink noise. The riffology algorithm makes dynamically weighted random choices for many parameters such as which riff from a repertoire of 32 eight-note melody fragments to play next, how fast to play it, how loud to play it, when to omit or elide notes, when to insert a rhythmic break, and other such choices. These choices are predicated on a model of a facile but unimaginative (and slightly lazy) guitarist. A few examples should illustrate the idea. To choose the next riff to play, the program selects a few possibilities randomly (the ones that "come to mind" in the model). From these it selects the riff that is "easiest" to play, i.e. the riff whose starting note is closest to one scale step away from the previous riff's ending note. To decide whether to skip a note in a riff (by replacing it with a rest or lengthening the previous note's duration) a dynamic probability is generated. That probability starts at a low value, rises to a peak near the middle of the solo, and drops back to a low value at the end. The effect is that solos start with a blur of notes, get a little lazy toward the middle and then pick up energy again for the ending. The solo is accompanied by a bass line, rhythm pattern, and chords which vary less randomly but with similar choices. The result is an infinite, non-repeating improvisation over a non-repeating, but soon familiar, accompaniment.

A version of "Song of the Grid" forms part of Eedie's telephone demo. This *finite* version has a more standard structure; the accompaniment has a fixed A-A-B-A pattern and a precomposed "head" (melody) is played the first time through the pattern. The following improvisation uses one, two, and finally three interdependent voices. The repertoire of riffs has been increased by about 40% and now contains some famous phrases from early jazz guitarists who, being dead, could not be asked for permission (the original selection contained many riffs contributed by friends and used with their consent).

Figure II is the beginning of one of the improvisations produced. The music generated by this algorithm passes the "is it music?" test; "Song of the Grid" makes perfectly acceptable background music, (better than that heard in most elevators or supermarkets) and even won lavish praise in reviews of "*ballblazer*" in national publications. However it doesn't pass the "is it interesting music?" test after the first ten minutes of close listening, because the rhythmic structure and the large scale melodic structure are boring. It appears that for music to remain *interesting* it must have appropriate structure on many levels.

Music Generated by Formal Grammars — L-Systems

In the nineteen-sixties, Aristid Lindenmayer proposed using parallel graph grammars to model growth in biological systems [LINDENMAYER68]; these grammars are often called "Lindenmayer-systems" or simply "L-systems". Alvy Ray Smith gives a description of L-systems and their application to computer imagery in his Siggraph paper on graftals [SMITH84].

Alphabet:	{a,b}
Axiom:	a
Rules:	a → b
	b → (a)[b]
Generation	String
0	a
1	b
2	(a)[b]
3	(b)[(a)[b]]
4	((a)[b])[(b)[(a)[b]]]
5	((b)[(a)[b]])[((a)[b])[(b)[(a)[b]]]]
6	(((a)[b])[(b)[(a)[b]]) [((b)[(a)[b]]) [((a)[b])[(b)[(a)[b]]]]]]

Figure III - "FIB", a simple bracketed OL-system

Figure III shows the components of a simple bracketed OL-system grammar⁴ and the first seven strings it generates. This example is one of the simplest grammars that can include two kinds of branching (e.g. to the left for '(' and to the right for '['). The name "FIB" was chosen for this grammar because the number of symbols (as and bs) in each generation grows as the fibonacci series.

L-systems exhibit several unusual characteristics and three are of particular interest here. The first is that they can be used to generate graphic entities that really look like plants. Although the grammars themselves are quite mechanistic and have none of the apparent randomness of natural phenomena, the graphic interpretations of the strings generated are quite believably "natural". The second characteristic is structural. The "words" of the grammar (the strings produced by repeated application of the replacement rules) have a fine structure defined by the most recent replacements and a gross structure defined by the early replacements. Since all the replacements are based on the same set of rules, the fine and gross structures are related and produce a loose form of self-similarity reminiscent of the so-called "fractals" (although none of the entities produced by graph grammars are fractal in the strict sense of the word). The third characteristic of L-systems is a property called "database amplification", the ability to generate objects of impressive complexity from simple sets of rules (i.e. generating a lot of output from little input). We would like any composition algorithm to be significantly less complicated than the music it generates.

The generation of the strings (or "words") in OL-systems is strictly defined; every possible replacement must be performed each generation, or, to put it another way, every symbol that can be replaced will be. This means that the process is deterministic as long as no two replacement rules have the same left side. Note that in cases where the rewriting rules specify replacing a single symbol with many replaceable symbols (the typical case), growth will be exponential.

Although generation of the strings is well defined, an additional *interpretation* step is required to express these strings as graphic or auditory entities. Figure IV shows a very simple graphic interpretation of several grammars, including that of FIB.⁵ In this interpretation the symbol '(' begins a branch at an angle of 22.5° to the "left" (i.e. counterclockwise) and the symbol '[' begins a branch at an angle of 28.125° to the "right" (i.e. clockwise). The examples are all scaled to make their heights approximately equal; the relative scaling ranges from 1.0 for FIB to 0.026 for

⁴ A *OL*-system is a context insensitive grammar; a *1L*-system includes the nearest neighbor in the context; a *2L*-system includes 2 neighbors; etc. A *bracketed* L-system is one in which bracketing characters (typically '[' and ']', or '(' and ')', or others) are added to the grammar as placeholders that indicate branching, but are not subject to replacement.

⁵ Missing from the grammar descriptions are their alphabets ({a,b}, {a,b}, {a,b,c}, {a,b,c,d,e,f,g}, and {a,b,c,e,f,g} respectively) and their axioms (a for all of them).

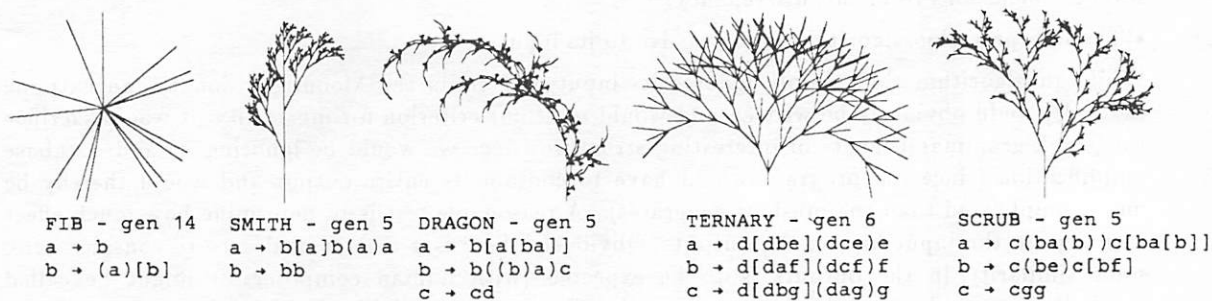


Figure IV - graphic interpretations of some graph grammars

SCRUB. In the first example (FIB) 716 separate line segments radiate from the starting point, many overlaid on each other. The resulting figure is 2 line segments tall. In the second example ("SMITH", borrowed from [SMITH84]) 665 line segments branch from each other and produce a figure that is approximately 60 line segments tall. The replacement rules for SMITH are more complicated than those for FIB (although still quite simple), but even when comparing strings of approximately equal length from the two grammars, the graphic interpretation of SMITH is qualitatively more complicated and "natural" looking than that of FIB. Although changing the interpretation scheme could elicit a more ornate graphic for FIB, (e.g. by avoiding the overlaid segments), the extreme simplicity of the underlying structure makes it impossible to achieve the kind of complexity that we expect in a "natural" object. However, adding a little more variety in the structure (as in SMITH) appears to be sufficient to generate that complexity.



Figure V - a musical interpretation of the grammar from figure III

The music in figure V is an interpretation of the seventh generation of FIB. The interpretation algorithm used for this example performed a depth-first traversal of the tree. At each left bracketing symbol $\{[, \{, ($ a variable "branch" was incremented. At each right bracketing symbol $\{], \},)$ "branch" was decremented. At each alphabet symbol $\{a, b\}$ a variable "seg" was incremented if the symbol was b, "branch" and "seg" were then used to select one of 50 sets of four notes, and "branch" was used to select one of 13 metric patterns that play some or all of the notes. This is one of the more complicated algorithms that were used.

The fragment in figure V is pleasing and demonstrates a reasonable amount of musical variety. This contrasts sharply with the undeniably boring graphic in figure IV. We could draw any one of

several conclusions from this discrepancy:

- The interpretation algorithm is insensitive to its input

While an algorithm that simply ignores its input and emits the Moonlight Sonata (an extreme example) could obviously be written and would meet our criterion for musicality, it would sacrifice the graph grammar benefits of interesting structure (since we would be ignoring it) and database amplification (since the program would have to contain its entire output and would thereby be more complicated than the music it generates). A reasonable test is to determine how much effect a change to the input has on the output. Obviously there is a matter of degree to consider here; some similarity in the outputs is to be expected (with human composers it might be called "style"), but we also expect easily recognized differences. When this algorithm is applied to other grammars, the output generated has recognizable similarities and differences, so we must conclude it is not insensitive to its input.

- Music requires less complexity than graphics.

Music clearly requires *different* structure characteristics than graphics; the roles of fine and gross structure are almost exactly reversed in them. Music is sequential; it is experienced as a one-dimensional phenomenon that only varies with time (ignoring binaural effects), whereas (static) graphics is holistic; it is experienced as a two-dimensional phenomenon that does not vary with time. This is an important structural difference. When exposed to a graphic work, the first impression the viewer receives is of the gross structure; further examination fills in the finer structure in a sequence determined by the viewer's eye motion.⁶ When exposed to a musical work, the first impression is of fine structure in one specific area (the first measure), followed by fine structure in the next area (the next measure), and so on. Only after perceiving the fine structure can the listener know the gross structure of a musical piece.

In his paper "The Complexity of Song" [KNUTH77], Donald Knuth proves the theorem "There exist arbitrarily long songs of complexity $O(1)$ " by showing that a song recorded by Casey and the Sunshine Band has a lyric consisting of eight words arranged in a fourteen word long pattern and repeated arbitrarily many times.⁷ Although Knuth's paper dealt with the *text* of songs (and was published as a joke), he touches on an important aspect of melodic structure; reiteration of previous segments. It is commonly accepted that expectation, based on the inductive/deductive process of guessing the overall structure and predicting what will come next, must be satisfied most, but not all, of the time in order for music to be pleasing. This can only be a consideration in a medium in which the structure is revealed sequentially. A simple way to achieve the middle ground between boring predictability and frustrating arbitrariness is to introduce a pattern and then repeat it with variations.

The common implication is that "high" music tends away from predictability and "low" music (as implied in Knuth's example) tends toward it. As one extreme, a long series of notes chosen randomly does not pass the "is it musical?" test. (Some very modern composers seem not to have noticed; is it because they are aiming at a very "high" audience?) Extreme complexity in music is perceived as arbitrariness, i.e. an excessively complex structure is perceived as no structure at all. Similarly, a screen full of many randomly colored pixels appears to be without structure and would fail the "is it a graphic?" test. On the other hand, a small number of randomly chosen notes or randomly colored blotches is often considered "musical" or "graphical". One possible explanation is that since many patterns will match a small part of a random sequence, the small sample of randomness is perceived as a small sample of a "real" (i.e. predictable) pattern. An interesting measure would be how big a random pattern can get before it is recognized as meaningless. A comparison of these measures for graphic and musical entities might help answer the

⁶ "Progressive" picture transmission techniques take advantage of this observation. Gross details are sent first, followed by successively finer details until the viewer is satisfied [KNOWLTON80] [FRANK80].

⁷ "That's the way, uh huh, uh huh, I like it, uh huh, uh huh, ..."

related question, "does music *tolerate* less complexity than graphics?"

- The structure in FIB is, for some reason, more appropriate for music than for graphics.

The requirement that a balance be maintained between the repetition of old material and the introduction of new material in music implies that structures that contain repeats or near-repeats of segments are particularly appropriate. Static graphics has no time dimension in which temporal repetition can occur, and spatial repetition, while not unknown, is not as important in graphics as temporal repetition is in music. Although the syntax of graphics does not require repetition, the syntax of plants (which is the *semantics* of figure IV) does require repetition. Simply stated, plants manifest a structure composed of varied repetitions, just as music typically does. Is this a case of art imitating nature?⁸ The structure in FIB, which is almost entirely repetition, satisfies one of the important requirements of music (and plants), but has little to offer for graphics per se.

192 samples of music were produced by trying twelve different interpretation algorithms on the third generation strings of each of 16 different grammars. The samples ranged from 0.0625 bars long (one note lasting 0.15 seconds) to 46.75 bars long (about 2 minutes). A small group of evaluators listened in randomized order to every sample and rated them on a 0 to 9 scale; 0 for "awful" through 3 for "almost musical" and 6 for "pleasing" to 9 for "wonderful". Of these 192 samples ~89% rated above 3.⁹ Some algorithms performed very well and generated not only "musical" but "pleasing" results on the average. Only one of the algorithms (the first one designed) averaged below 3. If that algorithm is discarded the success rate becomes ~95%.

Eddie plays several examples of melodies generated by L-system grammars in her demo, including one in which sequences derived from two different grammars are interpreted by the same algorithm and overlaid. The similarities resulting from the common interpretation scheme give the piece a fugal quality while the differences in the grammars cause the two melodies to diverge in the middle and (by luck) converge at the end.

Music Generated by "Stochastic Binary Subdivision" — DDM

The metric character of western popular music exhibits an extremely strong tendency. It is binary. Whole notes are divided into half notes, quarter notes, eighths, sixteenths, etc. And, while a division into three sometimes happens, as in waltzes or triplets, thirds are almost never subdivided into thirds again in the way that halves are halved again and again.¹⁰ Further, it is rare for notes started on "small" note boundaries to extend past "larger" note boundaries. More precisely, if we group the subdivisions of a measure into "levels", such that the n th level contains all the subdivisions which are at odd multiples of 2^{-n} into the measure (e.g. level 3 consists of the temporal locations $\{1/8, 3/8, 5/8, 7/8\}$), we find that notes started on level n infrequently extend across a level $n-1$ boundary and only rarely extend across a level $n-2$ boundary.

Rhythms, like melodies, must maintain a balance between the expected and the unexpected. As long as, *most of the time*, the primary stress is on the "one beat" with the secondary stress equally spaced between occurrences of the primary stress, and so forth, that balance is maintained. By making note length proportional to the level of subdivision of the note beginnings, emphasis is constantly returned to the primary stress cycle. Simple attempts at generating rhythms "randomly" fail because they ignore this requirement. More sophisticated attempts make special checks to avoid extending past the primary or the secondary stress, but the result is still erratic because we expect to hear music that keeps reiterating the simple binary subdivisions of the measure.

⁸ "For when there are no words it is very difficult to recognize the meaning of the harmony and rhythm, or to see that any worthy object is imitated by them." [PLATO55]

⁹ The median value was 5.0, but I would have been happy had it been 3.

¹⁰ Waltzes and triplets probably gain much of their effect from the feeling that a beat is missing from a pattern of four, or has been added to a pattern of two.

```

divvy(ip, lo, hi)
struct instr *ip;
int    lo, hi;
{
    int mid = (lo + hi) >> 1;

    ip->pat[lo] = 'I';
    if ((rand() % 100) < ip->density && hi - lo > ip->res) {
        divvy(ip, lo, mid);
        divvy(ip, mid, hi);
    }
}

```

Figure VI - Subroutine "divvy" from ddm.c

The program "ddm"¹¹ attempts to make musical rhythms by adhering to the observations made above, and making all other choices randomly. Figure VI is the heart of ddm. The structure `instr` contains, among other things, `density` - a probability that, at any level, subdivision to the next level will occur, `res` - the shortest note that can be generated, i.e. the deepest level to which it can subdivide, and `pat` - a character string in which the generated pattern of beats is

```

45:75: 2:0: 96:1 BD
52:75: 4:0: 96:1 SD
57:50: 8:0:120:1 HHC
51:50: 8:0: 64:1 RIM
48:67: 8:0: 80:1 TOM3
54:70: 8:0: 64:1 CLAP
55:75: 8:0: 64:1 COWB
59:50:16:0: 80:1 HHO
53:67:16:0: 72:1 TOM1

```

Figure VII - Typical Data File for DDM

stored. Figure VII shows a typical input file for the program. The first column is the code that must be sent to a drum machine to make the intended drum sound; the second column is the percent chance that subdivision will occur at any level; the third column is the smallest subdivision allowed; the fourth column is the maximum duration of a note (0 for drum machines); the fifth column is how loud the sound should be; and the sixth column is which MIDI channel (i.e. which drum machine) should play it. Any further information on the line is comment, in this case the instrument name.

Figure VIII shows the output of ddm in two forms; the first is one measure of debugging information showing the results of all subdivisions and the second is two bars of drum score showing the final result. Note that only one instrument is played at any one time; 'I' indicates a subdivision, and '#' indicates the drum to be played at that time ("pound" sign seemed appropriate, somehow). Precedence is given to the instruments listed earliest in the input file, thus the bass drum (BD) plays the down beat, even though all the instruments wanted to appear then; similarly, the low tom-tom (TOM3) plays on the next eighth note, having precedence over the open hi-hat (HHO). The drum score in figure VIII starts with the same measure and then goes on for another measure which is quite different although based on the same set of probabilities and precedences.

If we let the lines in the ddm file describe notes and pitch differences instead of drum types, ddm can be used to generate melodies. This is a simple change in the implementation; the drums are

¹¹ a.k.a. digital drum madness

A2	#.....#.....#.....	BD
E3#.....#.....	SD
A3#.....#.....	HHC
Eb3#.....#.....	RIM
C3#.....#.....#.....	TOM3
Gb3#.....#.....#.....	CLAP
G3#.....#.....#.....	COWB
B3#.....#.....#.....	HHO
F3#.....#.....#.....	TOM1



Figure VIII - Sample DDM Output

already encoded as pitches (45 \equiv A2 \equiv bass drum, 52 \equiv E3 \equiv snare drum, etc.). The only extension that is needed is to encode pitch differences. This we do by defining any value of 12 or smaller to be a pitch difference (by doing so, we make C#0 the lowest note we can specify directly). By adding lines like 1:60:16:31:64:0 and -1:65:16:31:64:0, meaning go up

Scale 1,2,4,7,9
Limits 48,84
69:33: 8:12:64:0 A4
64:33: 8:12:64:0 E4
1:60:16:28:64:0 up
-1:65:16:28:64:0 down
1:55:32: 4:64:0 up
-1:50:32: 4:64:0 down

Figure IX - DDM File for Scat

one step and down one step respectively, rising and falling motions can be included. Figure IX shows an eight line file used to generate melodies to be "sung" by a Dectalk speech synthesizer. The "Scale" and "Limits" lines constrain the program to stay within a particular musical scale (a simple pentatonic) and within a certain range (C3, an octave below middle C, through C6, two octaves above middle C). By specifying fairly short notes in the fourth column, the chance of rests



Figure X - Sample Scat Output

appearing (for the singer to breathe) is increased. Figure X is a sample of dlm output from the file

in figure IX. The program "scat.c" converts the MIDI format output of ddm into a sequence of Dectalk commands that produces scat singing. The syllables are generated by combining a randomly chosen consonant phoneme with a randomly chosen vowel phoneme; the result is usually humorous (and occasionally vulgar).

Eddie uses ddm twice during the telephone demo, once to compose a little scat sequence to sing for the caller and once at the end to compose a piece for bass, drums, piano, and clavinet called "Starchastic X", where X is the process id of Eddie's process. Although no formal testing has been done with ddm, informal testing, (e.g. "Well, how'd you like it?"), always elicits compliments with no apparent doubts as to its musicality.

Results & Summary

The telephone demo, despite its high "down time" (due in part to hardware frailties and in part to its dependence on two largely unrelated sets of experimental software), has been a great success. Eddie and Eddie have received over a thousand calls and have given as many as 60 demos a day to callers from 50 different area codes. Although no formal announcements of their demo has ever been made (prior to this paper), word of mouth has brought calls from Belgium, Canada, England, Holland, and even Texas.

Although enough experience has been gained with the music generation algorithms to draw some tentative conclusions, it should be stressed that this is still a "work in progress". From a quick perusal of the "further work" section it should be obvious that many ideas and experiments have yet to be tried, and that, undoubtedly, these tentative results will be modified and refined.

The riffology technique generates "musical" sounds; it is *sufficient* but gets boring with repetition. Greater rhythmic variety and some interesting overall structure are needed.

L-system grammars generate strings that have at least one important musical characteristic — varied repetition. It is not yet clear whether the relationship between their fine structure and their gross structure is a benefit for music. It is clear that the database amplification property is useful in that an arbitrarily long piece can be generated. The majority of cases tried generate "musical" sounds, allowing us to call this technique *sufficient*.

The binary subdivision technique works reliably. A tendency to produce fast, rising or falling sequences must be avoided by careful file specifications, otherwise almost comic sequences result (not unlike Chico Marx at the piano). Given that such sequences are avoided, its results are "musical", so it is *sufficient*.

With all three differing algorithms showing sufficiency we must conclude that none of the three is *necessary*. The interesting possibility remains that the overlap (intersection) of the three algorithms would be sufficient by itself. Unfortunately, the intersection of the three sets of rules is so trivial {use diatonic pitches, use no note shorter than a sixty-fourth note}, that it is unlikely that it is sufficient.

It is interesting to note that many of the pieces generated by these algorithms appear to have semantic content, (some seem to brood, some are happy and energetic, others bring Yuletide street scenes to mind). Since the algorithms themselves have no information about human emotions or the smell of chestnuts roasting on an open fire, we must conclude that any semantically significant constructions that occur are coincidental. Our semantic interpretation of these accidental occurrences, like seeing a man in the moon, testifies to the ability of human consciousness to find meaning in the meaningless.

Further Work

The work presented here just scratches the surface of a very complicated subject area. For every idea implemented during this project, ten were set aside for later consideration.

Among the ideas not yet tried for the "riffology" technique are:

- Expand the repertoire of riffs.
- Allow metric variation within the riffs.

- Enforce “breathing”, i.e. limit the length of phrases of rapid notes, inserting rests in the same way a singer must.
- Provide a mechanism for generating large-scale structure, perhaps by using grammar-driven or binary subdivision algorithms.

Among the ideas not yet tried for grammar based techniques are:

- Experiment with much more complex grammars.
- Establish “controls” by feeding the interpretation algorithms random input and constant input. Compare the output from the controls to that from the grammars.
- Interpret the strings as rhythmic entities rather than melodic.
- Interpret the strings in larger chunks instead of symbol by symbol; perhaps by treating pairs or triples of symbols as objects, or by treating entire “branches” as objects.
- Experiment with 1L-systems or other context-sensitive grammars.
- Define analogues in the audio domain for the graphic interpretation of the strings such that the sounds produced are, in some non-trivial sense, equivalent to the graphics produced.
- Choose an audio interpretation of the strings and design some grammars specifically to sound musical with that interpretation; does the graphic interpretation of those grammars communicate the same components (whatever they are) that made the sounds musical?

Among the ideas not yet tried for binary subdivision techniques are:

- Allow multiple simultaneous instruments or notes (currently only one note is played at a time).
- Allow the events being selected to be at a higher level than notes (e.g. entire phrases or repeats of previous output).
- Allow the events being selected to be at a lower level than notes (e.g. slurring of notes or vibrato).
- Allow the events being selected to have more global scope (e.g. key changes or tempo shifts).

Among the ideas not yet tried for demonstrating music over telephone lines are:

- Solicit listener evaluations through the touch tone pad.
- Allow callers to select types of music or specify other parameters that influence the music generated.
- Provide stereo output (via two telephone lines).
- Charge money.

Acknowledgements

Several people have been particularly helpful with this project. Gareth Loy of UCSD did the initial work with connecting the Sun to MIDI instruments, wrote the original *record* and *play* programs, and proved it can be done. Michael Hawley of the Droid Works extended Gareth’s work, provided a flexible programming environment on the Sun, and, best of all, gave me copies of all that. Brian Redman is responsible for the entire BerBell project and often went out of his way to allow me to do things with the phone system that no civilized person would allow. Stu Feldman (my boss) has provided encouragement, equipment, and proofreading. He has yet to suggest I consider a career in sanitation engineering.

Credit is also due to fellow musicians who were excited by the idea of computers improvising music and contributed riffs: Steve Cantor, Mike Cross, Marty Cutler, Charlie Keagle, David Levine, Lyle Mays, Pat Metheny, and Richie Shulberg.

References

- CHOWNING73 John Chowning, “The Synthesis of Complex Audio Spectra by Means of Frequency Modulation” *Journal of the Audio Engineering Society* vol. 21, no. 7, pp. 526–534
- FRANK80 Amalie J. Frank, J. D. Daniels, and Diane R. Unangst, “Progressive Image Transmission Using a Growth Geometry Coding” *Proceedings of the I.E.E.E.*, Special Issue on Digital Encoding of Graphics, vol. 68, no. 7, pp. 897–909 (July,

- 1980)
- HILLER70 Lejaren Hiller, "Music Composed with Computers – A Historical Survey", *The Computer and Music*, Cornell University Press, pp. 42–96 (1970)
- KNOWLTON80 Ken Knowlton, "Progressive Transmission of grey-scale & binary pictures by simple, efficient, and lossless encoding schemes" *Proceedings of the I.E.E.E.*, Special Issue on Digital Encoding of Graphics, vol. 68, no. 7, pp. 885–896 (July, 1980)
- KNUTH77 D. E. Knuth, "The Complexity of Songs" *SIGACT News* vol. 9, no. 2, pp. 17–24 (1977)
- LANGSTON85 P. S. Langston, "The Influence of Unix on the Development of Two Video Games", EUUG Spring '85 Conference Proceedings, (1985)
- LEVINE84 D. Levine & P.S. Langston, "*ballblazer*", (tm) Lucasfilm Ltd., video game for the Atari 800, & Commodore 64 home computers, (1984)
- LINDENMAYER68 Aristid Lindenmayer, "Mathematical Models for Cellular Interactions in Development, Parts I and II," *Journal of Theoretical Biology* 18, pp. 280-315 (1968)
- MIDI85 "MIDI 1.0 Detailed Specification", The International MIDI Association, 11857 Hartsook St., No. Hollywood, CA 91607, (818) 505-8964, (1985)
- MORGAN73 S. P. Morgan, "Minicomputers in Bell Laboratories Research", Bell Laboratories Record, vol. 5, no. 7 p 194–201 (1973).
- PLATO55 Plato, *Laws*, Book II, (ca. 355 B.C)
- REDMAN85 B. E. Redman, "Who Answers Your Phone in the Information Age?", Usenix Summer '85 Conference Proceedings, (1985)
- REDMAN86 B. E. Redman, "BerBell", Bell Communications Research Technical Memorandum (in preparation)
- SMITH84 Alvy Ray Smith, "Plants, Fractals, and Formal Languages" *Computer Graphics* Proceedings of the Siggraph '84 Conference, vol. 18, no. 3, pp. 1–10 (July 1984).

APPENDIX

The following list contains brief descriptions of programs used to manipulate MIDI data or for some other aspect of the telephone demo. Most of them (81%) were written during my work on the demo, (see acknowledgements for other authors), and most of them (83%) were actually used in one way or another for the demo. It is a testimonial to the good influence of UNIX and the advocates of tool-making that fewer than a third of the programs are specific to the demo (3%), or specific to the composition algorithms (29%). Missing from this list are the (many) programs that operate our telephone switch and provide user access to it.

- 2332probe – Used by demo2332 to check pud's status
- adjust – perform metric adjustment dynamically
- atox – convert hexadecimal ascii to binary
- bars – count & select bars of MIDI data
- bbriffs – generate improvisation for "Song of the Grid"
- ccc – convert ascii chord charts to MIDI data
- chart – display midi data in PRN (piano roll notation)
- ched – interactive midi data editor (CHart Editor)
- chmap – select and remap MIDI channel events
- cntl – generate MIDI control change commands
- da – disassemble MIDI data to ASCII listing
- dack – lint for ASCII MIDI listings (da check)
- ddm – stochastic binary subdivision generator

decsquawk	- control a Dectalk voice synthesizer
harm	- harmonize melody lines (MIDI)
inst	- generate MIDI voice change commands
just	- adjust (quantize) note timings (MIDI)
keyvel	- scale, compress, and expand key velocity dynamics
kmap	- remap key values (pitches)
m2midi	- convert m-format scores to MIDI
m2p	- convert m-format scores to PIC macros for typesetting
mc	- C compiler with MIDI libraries
mdemo	- introduce, compose, and play the telephone demo
mecho	- add echo to MIDI data
merge	- combine MIDI data streams
metro	- metronome with graphic interface
midi2m	- convert MIDI data to m-format scores
midimode	- remove running status from MIDI data streams
mpuclean	- insert running status, remove other junk from MIDI data
mundef	- PIC macros for typesetting music scores
muzak	- interpret unsuspecting ASCII text as music
notedur	- change note lengths without changing tempo
p01a	- interpret 0L-systems musically by one method
p01b	- interpret 0L-systems musically by another method
.	- .
.	- .
.	- .
p011	- interpret 0L-systems musically by yet another method
p01pic	- interpret 0L-systems graphically for typesetters
play	- play (and overdub) MIDI data streams
punk	- generate a "soft-punk" melody and accompaniment
ra	- assemble (re-assemble) ASCII data into MIDI
reclock	- regenerate timing commands for MIDI data
record	- record MIDI input
rxkey	- print information about the Yamaha RX/11/15/21 key mappings
scat	- convert MIDI data into vocal scat for the Dectalk
select	- filter MIDI data by various criteria
stretch	- retime MIDI data
transpose	- transpose MIDI key event (pitch) data
trim	- remove "dead air" from MIDI data
tshift	- shift MIDI data temporally
unjust	- add slight random time variations to MIDI data
xtoa	- convert binary data to hexadecimal ASCII

Secure Networking in the Sun Environment

Bradley Taylor
David Goldberg
Sun Microsystems, Inc.

ABSTRACT

We present an authentication system that greatly improves the security of Sun's network environment. The system uses DES encryption and public key cryptography to authenticate both users and machines in the network. The system is general enough to be used by other UNIX[†] and non-UNIX systems.

Introduction

Sun's Remote Procedure Call (RPC) mechanism has proved to be a very powerful primitive for building network services. The most well-known of these services is the Network File System (NFS), a service that provides transparent file-sharing between heterogeneous machine architectures and operating systems. The NFS is not without its shortcomings, however. Currently, an NFS server authenticates a file request by authenticating the *machine* making the request, not the *user*. On a system accessing files with the NFS, it is a simple matter of running the *su* command to impersonate the rightful owner of the file. But the security weaknesses of the NFS are nothing new. The familiar command *rlogin* is subject to exactly the same attacks as the NFS because it uses the same kind of authentication.

A common solution to network security problems is to leave it the solution up to each application. A far better solution is to put the authentication at the RPC level. The result is a standard authentication system that can be used by all RPC-based applications, such as the NFS, secure logins and the Yellow Pages (a name-lookup service).

Our system allows us to authenticate users as well as machines. The advantage of this is that it gives the network environment an appearance similar to the familiar time-sharing environment. Users are not tied to machines for security purposes, just as they are not tied to terminals. They can login to any machine, and their login password is their passport to network security. No knowledge of the underlying authentication system is required. Our goal is a system that is as secure and easy to use as a time-sharing system.

Our assumptions are that any machine is capable of injecting arbitrary data into the network and picking up any data on it. We also assume that no machine is capable of packet smashing, that is, capturing packets before they reach their destination, changing the contents, and then sending it back on its original course. The attacks we consider dangerous are those involving the injection of data, such as trying to impersonate somebody by generating the right packets, or recording conversations and then replaying them later. We do not worry about passive eavesdroppers who merely listen to the network traffic, but are unable to impersonate anybody by doing so.

[†] UNIX is a trademark of AT&T

wishes to talk to a server, it generates at random a key to be used for encrypting the timestamps (among other things). This key is known as the *conversation key*. The client encrypts the conversation key using a public key scheme and sends it to the server in its first transaction. This key is the only thing that is ever encrypted using public key cryptography. The particular scheme we use will be described further on in this document. Suffice it to say for now that for any two agents A and B, there is a DES key K_{AB} which only A and B can deduce. This key is referred to from here on as the *common key*.

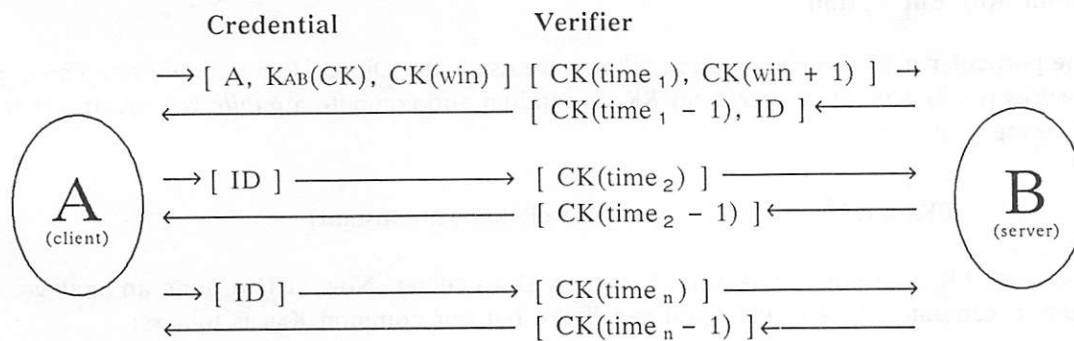


Figure 1: DES Authentication Protocol

Figure 1 above illustrates the authentication protocol in more detail, describing a client named A talking to server B. A term of the form $K(X)$ means X encrypted with the DES key K . Examining the table, you can see that for its first request, the client's credential contains three things: its name A, the conversation key CK encrypted with the common key K_{AB} and a thing called the *window* encrypted with CK. What the window says, in effect, to the server is this:

I will be sending you many credentials in the future, but there may be fiends sending them too, trying to impersonate me with bogus timestamps. When you receive a timestamp, check to see if your current time is somewhere between the timestamp and the timestamp plus the window. If it's not, please be so kind as to reject the credential.

The client's verifier in the first request contains the encrypted timestamp and an encrypted verifier of the specified window, $win + 1$. The reason this exists is the following. Suppose somebody wanted to impersonate A by writing a program that instead of filling in the encrypted fields of the credential and verifier, just stuffs in random bits. The server will decrypt CK into some random DES key, and use it to decrypt the window and the timestamp. These will just end up to be random numbers. After a few thousand trials, there is a good chance that the random window/timestamp pair will pass the authentication system. The window verifier makes guessing the right credential much more difficult.

After authenticating the client, the server stores four things into a credential table: the client's name A, the conversation key CK, the window, and the timestamp. The reason the server stores the first three things should be clear: it needs them for future use. The reason for storing the timestamp as well is to protect against replays. The server will only accept timestamps that are chronologically greater than the last one seen, so any replayed transactions are guaranteed to be rejected. The server returns to the client in its verifier an

index *ID* into its credential table, plus the client's timestamp minus one, encrypted by CK. The client knows that only the server could have sent such a verifier, since only the server knows what timestamp the client sent. The reason for subtracting one from it is to insure that it is invalid and cannot be reused as a client verifier.

The first transaction is rather complicated, but after this things go very smoothly. The client just sends its ID and an encrypted timestamp to the server, and the server sends back the client's timestamp minus one, encrypted by CK.

Public Key Encryption

The particular public key encryption scheme we use is the Diffie-Hellman method. The way it works is you generate a *secret key* SK_A at random and compute a *public key* PK_A using the following formula:

$$PK_A = \alpha^{SK_A} \quad (\alpha \text{ is a well-known constant})$$

You store PK_A in a public directory, but keep SK_A a secret. Now, if I've done an analogous thing to generate SK_B and PK_B , you can figure out our common K_{AB} as follows:

$$K_{AB} = PK_B^{SK_A} = (\alpha^{SK_B})^{SK_A} = \alpha^{(SK_A SK_B)}$$

Without knowing your secret key, I can calculate the same K_{AB} in a different way as follows:

$$K_{AB} = PK_A^{SK_B} = (\alpha^{SK_A})^{SK_B} = \alpha^{(SK_A SK_B)}$$

Notice that nobody else but the two of us can calculate K_{AB} , since doing so requires knowing either my secret key or yours. All of this arithmetic is actually computed modulo M , which is another well-known constant. It would seem at first that one could guess your secret key by taking the logarithm of your public one, but we choose M to be so large as to make this a computationally infeasible task. To be secure, K_{AB} will have too many bits to be used as a DES key, so what we do is extract 56 bits from it to be used as the DES key.

Both the public and the secret keys are stored in a public database indexed by netname, but the secret key is DES encrypted with your login password. When you login to a machine, the login program grabs your encrypted secret key, decrypts it with your login password, and gives it to a secure local keyserver to save for use in future RPC transactions. Note that ordinary users do not have to be aware of their public and secret keys. In addition to changing your login password, the *passwd* program will randomly generate a new public/secret key pair as well.

The keyserver is an RPC service local to each machine that performs all of the public key operations, of which there are only three. They are:

```
setsecretkey(secretkey)
encryptsessionkey(servername, deskey)
decryptsessionkey(clientname, deskey)
```

setsecretkey tells the keyserver to store away your secret key (SK_A) for future use and is

normally called by *login*. **encryptsessionkey** is called by the client program to generate the encrypted conversation key that is passed in the first RPC transaction to a server. The keyserver looks up **servername**'s public key and combines it with the client's secret key (set up by a previous **setsecretkey** call) to generate the key that encrypts **deskey**. The server asks the keyserver to decrypt the conversation key by calling **decryptsessionkey**. Note that implicit in these procedures is the name of caller, who must be authenticated in some manner. The keyserver cannot use DES authentication to do this since it would create deadlock. The keyserver solves this problem by storing the secret keys by uid, and only granting requests to local root processes. The client process then executes a set-uid process, owned by root, which makes the request on the part of the client, telling the keyserver the real uid of the client. Ideally, the three operations described above would be system calls, and the kernel would talk to the keyserver directly, instead of executing the set-uid program.

Naming

The old Unix authentication system has a few problems when it comes to naming. Recall that with Unix authentication, the name of a network entity is basically the uid. These uids are assigned per Yellow Pages naming domain, which typically spans several machines. We have already stated one problem with this system, that it is too UNIX oriented, but there are two other problems as well. One is the problem of uid clashes when domains are linked together. The other problem is that the superuser (uid 0) should not be assigned on a per domain basis, but rather on a per machine basis. The NFS deals with this latter problem in a severe manner: it does not allow superuser access over the network by uid 0 at all.

DES authentication corrects these problems by basing naming upon new names that we call *netnames*. Simply put, a netname is just a string of printable characters and fundamentally, it is really these netnames that we authenticate. The public and secret keys are stored on a per netname, rather than per username, basis. There is also a Yellow Pages map that maps the netname into a local uid and group-access-list, though other non-Sun environments may map the netname into something else.

The internet naming problem is solved by choosing the netnames to be globally unique. This is a far easier task than choosing globally unique uids. In the Sun environment, user names are assigned per Yellow Page domain. The convention we use for assigning netnames is to concatenate the username with the domain name. For example, a user named *dave* in the domain *discovery* would have the netname *dave@discovery*. If domain names are uniquely chosen within the internet, then this scheme will work to produce a unique netname for each user in the internet. A good convention for naming domains would be to append the ARPA domain name to the local domain name. Thus, the domain *discovery* within the ARPA domain *sun.com* would be renamed *discovery.sun.com*.

We solve the multiple superusers per domain problem by assigning netnames to machines as well as to users. A machine's netname is formed in a similar manner to a user's. Example: *dave's* machine has the netname *hal!root@discovery.sun.com*. Authenticating machines is a very important capability for diskless machines that need full access to their root filesystem over the net.

Other non-Sun environments will have other ways of generating netnames, but this does not preclude them from accessing the secure network services of the Sun environment. To authenticate users from any remote domain, all that has to be done is make entries for them

in two Yellow Pages databases. One is an entry for their public and secret keys, the other is for their local uid and group-access-list mapping. Upon doing this, users in the remote domain will be able access all of the local network services, such as the NFS and remote logins.

Applications of DES Authentication

We have only developed one application using DES authentication at the time of this writing, but we have two more planned. The first application we wrote was a generalized Yellow Pages update service. This service allows users to update their private fields in the Yellow Pages databases, for example, their login password or mail alias. Without the update service, we currently have a specialized daemon to update login passwords and hire a full-time person just to update the mail aliases! There are other applications for Yellow Pages updating, such as changing your login shell or adding a new machine to the host tables. No doubt many more applications will emerge.

Another application we have planned is a secure login program which we call *slogin*. *slogin* is aimed at being a replacement for *rlogin*, which suffers from some important security problems. *rlogin* operates in two modes: trusted and untrusted. In untrusted mode, the remote machine prompts you for a password, but then you are forced to send your password in the clear over the net. If the machine has an entry for your machine in its *hosts.equiv* file, then you can login in trusted mode, and there is no need to type in a password. But it is not that easy to tell which machines a given machine trusts, as you have to go recursively through the *hosts.equiv* file to determine this. It is important to emphasize that anybody able to change their uid on one of these trusted machines can login using your account, without typing a password. The fundamental problem with *rlogin* though is it should not be basing its security on machines, but rather on users as that is the way the *passwd* file is set up. If a machine trusts you, you should be able to login to it from any other machine, without having to send your password in the clear. *slogin* will correct these problems because the authenticated entity will be the netname instead of the machine-name. You should never be forced to type your password since you have already been authenticated. However, if you don't type your password, then the remote machine will not be able to decrypt your secret key, and you will not be able to access the secure network services from the remote machine. To correct this problem, there will be an option to *slogin* to force the remote machine to prompt you for a password. There will also be an option to *su* to prompt you for your password and set your secret key accordingly, in case you logged in without typing a password.

The other application that we have planned is the most important: the Network File System. There are three security problems with the current NFS using Unix authentication. The first is that verification of credentials occurs only at mount time when the client gets from the server a piece of information that is its key to all further requests called the *file handle*. Security can be broken if one can figure out a file handle without contacting the server, perhaps by tapping into the net or by guessing. After one has mounted an NFS file system, there is no checking of credentials during file requests and this brings up the second problem: if one has mounted a file system from a server that serves many clients (as is typically the case), then there is no protection against someone who has sovereignty over their machine using *su* (or some other means of changing the uid) to gain unauthorized access to other people's files. The third problem with the NFS is the severe method it uses to circumvent the problem of not being able to authenticate remote client superusers, that is, denying them

superuser access altogether.

The new authentication system will correct all of these problems. Guessing file handles is no longer a problem since in order to gain unauthorized access, the miscreant will also have guess the right encrypted timestamp to place in the credential, which is a virtually impossible task. The problem of authenticating root users is solved since we can now authenticate machines. The NFS server will associate with each file system that it serves a root netname which is granted full access to the filesystem. This is very important for diskless booting, so that an NFS server serving root filesystems will know which machine is associated with each root filesystem.

Actually, the level of security associated with each filesystem will be a user settable item. The file */etc/exports* currently contains a list of file systems and which machines may mount them. It will be changed in the future so that one can also specify the security required for accessing the filesystem. Unix authentication will be the default, but DES authentication can be selected as well. Associated with DES authentication is one parameter: the maximum window size that the server is willing to accept. There will also be a level of security where the actual file data is encrypted, if one is willing to accept the severe performance penalty for doing this.

Security Issues

There are several ways to break DES authentication, but we should make it clear first that using *su* is not one of them. The reason is the following. In order to be authenticated, your secret key must be stored by your workstation. This usually occurs when you login, with the login program decrypting your secret key with your login password and storing it away for you. If somebody tries to use *su* to impersonate you, it won't work because they won't be able to decrypt your secret key. Editing */etc/passwd* isn't going to help them out either, because the thing that they need to edit, your encrypted secret key, is stored in the Yellow Pages. If you log into somebody else's workstation and type in your password, then your secret key would be stored in their workstation and they could use *su* to impersonate you. But this is not a problem since you should not be giving away your password to a machine you don't trust anyway. Someone could just as easily change *login* to save all the passwords it sees into a file.

Not having *su* to rely on anymore, how is one to impersonate others in the new system? Probably the easiest way is to guess somebody's password, since most people don't choose very secure passwords. We offer no protection against this; it is your own problem if you choose an insecure password. The next way would be to attempt replays. For example, let's say that I have been squirreling away all of your NFS transactions with a particular server. As long as the server remains up, I won't succeed by replaying them since the server always demands timestamps that are greater than the previous ones seen. But now suppose I go and pull the plug on your server, causing it to crash. As it reboots, its credential table will be clean, and so it has lost all track of previously seen timestamps and now I am free to replay your transactions. There are few things to be said about this. First of all, servers should be kept in a secure place so that no one like myself will go and pull the plug on them. But even if they are physically secure, servers still occasionally crash without any help. Replay transactions is not a very big security problem, but even so, there is protection against it. If a client specifies a window size that is smaller than the time that it takes a server to reboot (5 to 10 minutes), then the server will reject any replayed transactions because they will have

expired. There are other ways to break DES authentication, but they are much more difficult. These methods involve breaking the DES key itself, or computing the logarithm of the public key. But it is important to keep our goals in mind. We are not aiming at having super-secure computing. What we're after is something that is as good as a time-sharing system, and in that end, we have been successful.

There is another security issue that DES authentication does not address, and that is tapping of the net. Even with DES authentication in place, there is no protection against somebody merely watching what goes across the net. This is not a big problem for most things, such as the NFS, since very few files are not publically readable, and besides, trying to make sense of all the bits flying over the net is not a trivial task. For logins, this is a bit of a problem because you wouldn't want somebody to pick up your password over the net. For this reason, the new remote login program *slogin* will encrypt all data (though there will be a way to turn this feature off). As we mentioned before, a side effect of the authentication system is a key exchange, so that the network tapping problem can be tackled on a per application basis.

Performance

Public key systems are known to be slow, but there is not much actual public key encryption going on in our system. Public key encryption only occurs in the first transaction with a service, and even then, there is caching that speeds things up considerably. The first time a client program contacts a server, both it and the server will have to calculate the common key. The time it takes to compute the common key is basically the time it takes to compute an exponential modulo M . On a Sun-3 using a 128-bit modulus, this takes roughly 1 second, which means it takes 2 seconds just to get things started since both client and server have to perform this operation. This is a long time, but it is only the very first time you contact a machine that you'll have to wait. Since the keyserver caches the results of previous computations, it does not have to recompute the exponential every time.

The most important service in terms of performance is the NFS, but at the time of this writing, we have not built the secure NFS so there is not much to be said about it. What we can give you are estimates. The extra overhead that DES authentication creates versus Unix authentication is most likely the encryption. A timestamp is a 64-bit quantity, which also happens to be the DES block size. Four encryption operations take place in an average RPC transaction: client encrypts the request timestamp, server decrypts it, server encrypts reply timestamp and client decrypts it. On a Sun-3, the time it takes to encrypt one block is about half a millisecond if performed by hardware and 1.2 milliseconds by software. So, the extra time added to the round trip time is about 2 milliseconds for hardware encryption and 5 for software. The round trip time for the average NFS request is about 20 milliseconds, resulting in a performance hit of 10 percent if one has encryption hardware, and 25 percent if not. Remember that is the impact on *network* performance. The fact is that not all file operations go over the wire, so the impact on total system performance will actually be lower than this. It is also important to remember that security is optional, so environments that don't require it may turn it off if they want higher performance.

Problems: Booting and Set-uid Programs

Consider the problem of a machine rebooting, say after a power failure at some strange hour when nobody is around. All of the secret keys that were stored away get wiped out, and now no process will be able to access secure network services, such as trying to mount an NFS

filesystem. The important processes at this time are usually root processes, and so if only *root's* secret key was stored away, then things would work okay, but nobody is around to type the password that decrypts it. One solution to this problem is to store the root password in a file, which the keyserver then reads to decrypt the secret key. This works fine for diskful machines which can store the root password on a physically secure local disk. It won't work for diskless machines though, since the root password must be sent in the clear over the net. Another solution is to mount all the filesystems read-only at boot-time, and then wait for the root user to return. After the root password has been entered, the filesystems can be remounted with full access rights. We will break all of the programs that depend upon writing files at this time, most notably administration and mail. The programs will have to be fixed on a per application basis. For example, mail can be sent to server machines instead of clients. As an aside, note that storing mail on the server has the windfall benefit of allowing one to read mail from several different machines, further reducing the workstation to user binding.

Another booting problem is the single-user boot. There is a mode of booting known as single-user mode, where one is given a *root* login shell on the console. The problem here is that one is not prompted for a password when attempting to do this. To fix this, in the future a password will be required in order to boot single-user. There will still be a way to set up your machine so that you can do single-user booting without a password, but you had better have physical security for your machine if you're going to turn this feature off.

Yet another problem with booting is that for diskless machines it will not be totally secure. It is possible for somebody to impersonate the boot-server, and boot you a devious kernel that, for example, makes a record of your secret key on a remote machine. The problem is that our system is set up to provide protection only after the kernel and the keyserver are running. Before that, there is no way to authenticate the replies given by the boot server. We don't consider this a serious problem because it is highly unlikely that somebody would be able to write this funny kernel without source code. Also, the crime is not without evidence. If you polled the net for boot-servers, you would discover the devious boot-server's location.

Set-uid programs will not behave as they should in all cases. If a set-uid program is owned by *dave*, and *dave* has not logged into the machine since it booted, then the program will not be able to access any secure network services as *dave*. The good news is that most set-uid programs are owned by *root*, and since *root's* secret key is always stored at boot time, these programs will behave as they always have.

Conclusion

Recall that our goal was a system that is as secure as time-sharing. We feel we have met this goal. The way you are authenticated in a time-sharing system is by knowing your password. In our system, the same is true. In time-sharing the person you trust is your system administrator, who does not do anything dirty such as change your *passwd* entry so they can impersonate you. In our system, you instead trust your network administrator who does not change your entry in the public key database. In one sense, our system is even more secure than time-sharing. In our system, it is unfruitful to place a tap on the network in the hopes of catching a password or encryption key, because we encrypt them. Most time-sharing environments do not encrypt the data emanating from the terminal; users must trust that nobody is tapping their terminal lines.

DES authentication is not the end-all authentication system for Sun. It is likely that in the future there will be sufficient advances in algorithms and hardware to render the public key system as we have defined it useless. The nice thing about DES authentication is that there is a smooth migration path for it in the future. Syntactically speaking, nothing in the protocol requires the encryption of the conversation key to be Diffie-Hellman, or even public key encryption in general. To make the authentication stronger in the future, all that needs to be done is to strengthen the way the conversation key is encrypted. Semantically though, this will be a different protocol, but the beauty of RPC is that it can be plugged in and live peacefully with the older authentication systems.

But for the present at least, DES authentication satisfies our requirements for a secure networking environment. From it, we are able to build a system secure enough for use in unfriendly networks, such as for example a student-run university workstation environment. The price for this security is not high. Nobody has to carry around a magnetic card or remember any hundred digit numbers. You use your login password to authenticate yourself, just as you did before. There is a small impact on performance, but if this worries you and you have a friendly net, you can merely turn the authentication off.

Acknowledgements

David Goldberg was the chief designer of DES authentication as well as how it fits in with the Sun environment. Bradley Taylor did most of the implementation, some of the design and even wrote this paper. We would like to thank Bob Lyon for putting authentication at the RPC level where it belongs, and for bringing RPC to Sun in the first place.

References

Diffie and Hellman, "New Directions in Cryptography", *IEEE Transactions on Information Theory* IT-22, November 1976.

Gusella & Zatti, "TEMPO: A Network Time Controller for a Distributed Berkeley Unix System", *USENIX 1984 Summer Conference Proceedings*, June 1984.

National Bureau of Standards, "Data Encryption Standard", *Federal Information Processing Standards Publication 46*, January 15, 1977.

Needham & Schroeder, "Using Encryption for Authentication in Large Networks of Computers", *Xerox Corporation CSL-78-4*, September 1978.

RPC Authentication

RPC is at the core of our security system, and in order to understand the big picture, we must first delve down to this level and understand how authentication works in RPC. RPC's authentication is open-ended: a variety of authentication systems may be plugged into it and coexist in the network. Currently, we have two: *Unix* and *DES*. DES authentication is the new system which is the main focus of this paper, while Unix authentication is the older, weaker system. Two terms are used when speaking of any RPC authentication system: *credentials* and *verifiers*. Using ID badges as an example, the credential is what identifies a person: his or her name, address, birthdate, etc. The verifier is the photo attached to the badge, so that one can be sure that the badge has not been stolen by checking the photo on the badge against the person carrying it. In RPC, things are much the same. The client process sends both a credential and a verifier to the server with each RPC request. The server sends back only a verifier, since the client already knows the server's credentials.

Unix authentication is the authentication system used by most of Sun's network services today. The credentials contain the client's machine-name, uid, gid and group-access-list. The verifier contains nothing. There are two problems with this system. The glaring one is the empty verifier! It is easy to use *hostname* and *su* to cook up just the right credential. If you trust all of your root users in the network, then this is not really a problem. But many times this is an unrealistic assumption. The NFS tries to combat the deficiencies in Unix authentication by checking the source internet address of mount request as a verifier of the hostname field, and accepting requests only from privileged internet ports. Still, it is not difficult to circumvent these measures, and NFS has no way of verifying the uid field.

The other problem with Unix authentication appears right in its name, *Unix*. It is another unrealistic assumption that all of the machines in your network will be UNIX machines. The NFS in fact works on MS-DOS and VMS machines, but Unix authentication breaks down considerably when applied to them. Given these two shortcomings, it is clear what one would desire in a new authentication system: operating system independent credentials and secure verifiers. This is the essence of DES authentication.

DES Authentication

The security of DES authentication is based upon a sender's ability to encrypt the current time, which the receiver can then decrypt and check against its own clock. The method used to encrypt this timestamp is DES (Data Encryption Standard). Two things must be true in order for this scheme to work: (1) the two agents must agree on what the current time is, and (2) the sender and receiver must be using the same encryption key.

If one has a network time synchronization, such as Berkeley's TEMPO, then client/server time synchronization is not a problem as it is performed automatically. However, if this does not exist, the timestamps can all be computed using the server's time instead of network time. In order to do this, the client asks the server for the time before starting the RPC session and computes the time difference between his own clock and the server's. This information is then used to offset the client's clock when computing timestamps. If the client and server's clocks get out of sync to the point where the server begins rejecting the client's requests, the DES authentication system will just resynchronize with the server again.

We now describe how the client and server arrive at the same encryption key. When a client

A Framework for Networking in System V

*David J. Olander
Gilbert J. McGrath
Robert K. Israel*

AT&T
190 River Road
Summit, NJ 07901

ABSTRACT

UNIX* System V was typified by the lack of a consistent framework for implementing networking services. This resulted in a collection of incompatible, inconsistent network protocols and utilities implemented under the traditional character I/O sub-system. Ritchie's Stream Input-Output system^[1] (STREAMS) has provided the much needed structure and modularity to this part of the operating system.

A new framework has been designed to support the development of network protocols and services in System V. The framework is based on STREAMS, and includes a set of service interfaces at critical protocol layers. The expected benefit of this framework is a consolidated set of user-oriented network services, as well as a structured environment for architecting protocols within the kernel.

1. Introduction

UNIX System V has been criticized in the past for its lack of support of networking services. Previous networking packages have been developed in an ad hoc manner using the traditional character I/O sub-system. Each package typically defined its own interface to available networking services and its own family of protocols, thereby creating a hodgepodge of incompatible, inconsistent network protocols and utilities available with System V. This has resulted in confusion for users of network services and an increasing collection of redundant software. There was no clear strategy for consolidating UNIX system networking services over a growing set of protocol suites (including TCP/IP, ISO, SNA) and network media.

An attempt to bring order to System V network development has resulted in a new framework for networking. The core of the framework is the definition of protocol service interfaces modeled after the Reference Model of Open Systems Interconnection (OSI)^[2]. The networking framework defines these interfaces in the context of Ritchie's Stream Input-Output system^[1] (hereafter referred to as STREAMS). STREAMS provides a new structure to the character I/O sub-system that encourages the modular software structure typified by layered protocol architectures. In addition, STREAMS provides the mechanism that enables one to connect applications and protocols with common service interfaces.

This paper describes the structure and benefits of the new networking framework. It avoids a full discussion of the STREAMS mechanism, which has previously been described. First, the goals of the framework are discussed. An abstract representation of the framework is then presented. Finally, the mechanism for defining networking service interfaces using STREAMS is presented, and existing service interfaces are described.

* UNIX is a trademark of AT&T.

2. Networking Goals

Three objectives were defined for supporting networking services in System V. The first is the consolidation of user-oriented services. A standard set of networking services, such as file transfer, should be provided with the UNIX system. These services should be offered regardless of the underlying protocols or network media that may be available between two systems. Furthermore, a single implementation of each service should operate in any protocol or media environment.

The second goal is to provide kernel support for modular protocols. Contemporary protocol architectures and the OSI Reference Model evidence the industry consensus for modularizing protocol functions. Benefits of a modular structure include the ability to isolate a higher layer protocol or application from low layer protocols, thereby realizing the transparent substitution of protocols beneath a module.

The third goal of our framework is to flexibly support a growing set of protocol families, including TCP/IP, ISO, XNS, and SNA.

3. The Abstract Solution

Before getting bogged down in details of the framework implementation, it may be useful to understand an abstract view of the framework. Later, we will discuss the specific services that have been provided within the kernel and at user level.

The three objectives stated earlier can be realized using two basic principles of the OSI Reference Model: *functional layering* and *service interfaces*. Functional layering is used to simplify the design complexity of network systems. A network system is divided into layers, where each layer provides a set of services to the next higher layer and shields that layer from details of how its services are implemented.

The service interface is the boundary between two layers (see Figure 1). It defines the services offered by the lower layer to the higher layer. A service interface consists of the set of primitives that provide specific services, plus the rules for using those primitives (state transition rules). A service primitive may be a user request for a particular service, or an indication of a pending event.

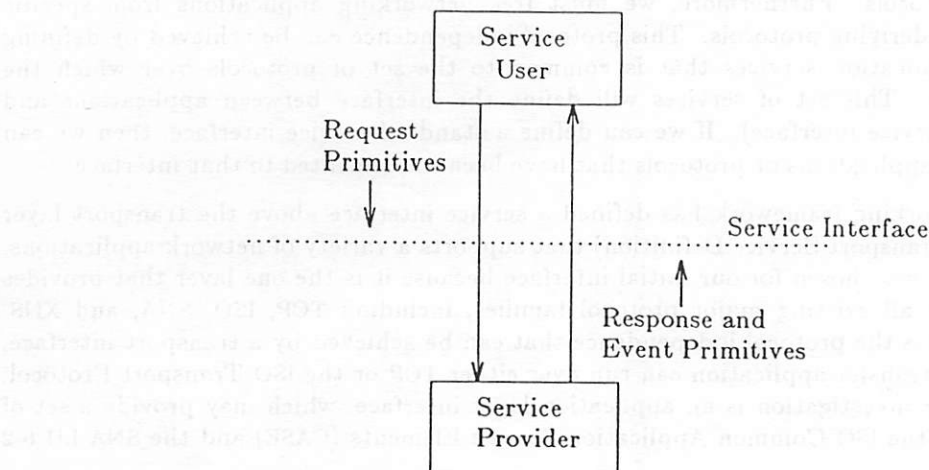


Figure 1. Service Interface

Figure 2 presents a concrete example of a service interface. It shows the state transition diagram for the ISO Transport Service Definition^[3]. The diagram specifies the legal states, and the transitions that occur when primitives are generated from each state. For example, a user may request the establishment of a transport connection by passing a T-CONNECT-request

primitive to the service provider. When the provider returns a notification that the connection has been established (T-CONNECT-confirm primitive), the data transfer state is entered and communication may begin.

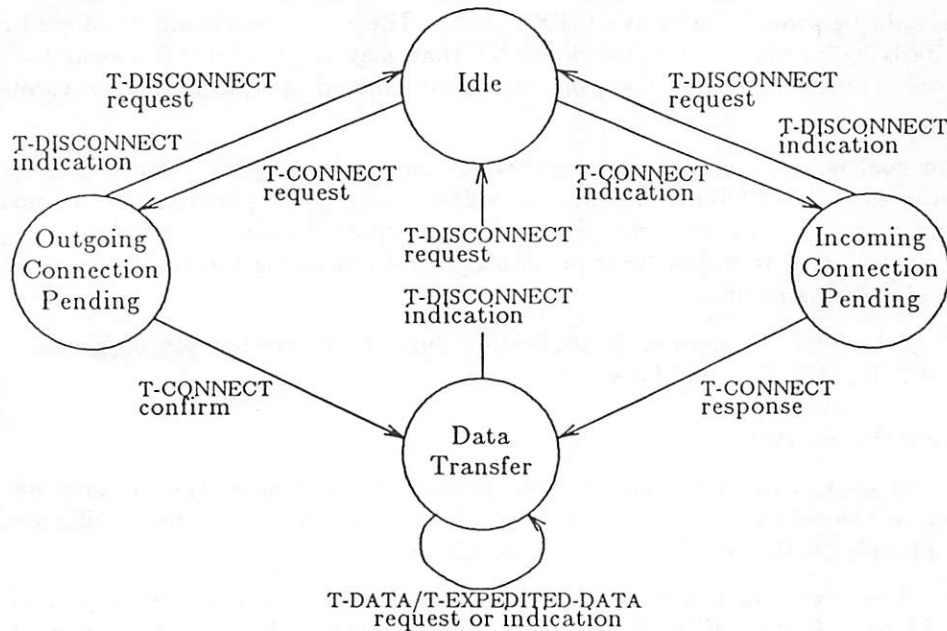


Figure 2. Transport State Transition Diagram

Below is a description of how functional layering and services interfaces can be used to satisfy each objective of our framework.

3.1 Consolidating Network Services

As discussed earlier, we would like to provide a standard set of networking services with System V that operate over any protocol or network medium available with a system. Functional layering will enable us to separate the networking application from the underlying communication protocols. Furthermore, we must free networking applications from specific knowledge of the underlying protocols. This protocol independence can be achieved by defining a subset of communication services that is common to the set of protocols over which the application will run. This set of services will define the interface between applications and protocols (i.e. the service interface). If we can define a standard service interface, then we can connect any pair of applications or protocols that have been implemented to that interface.

The System V networking framework has defined a service interface above the transport layer (based on the ISO Transport Service Definition) that supports a variety of network applications. The transport layer was chosen for our initial interface because it is the one layer that provides common services to all existing major protocol families, including TCP, ISO, SNA, and XNS. Figure 3 demonstrates the protocol independence that can be achieved by a transport interface, where the same file transfer application can run over either TCP or the ISO Transport Protocol. Another layer under investigation is an application layer interface, which may provide a set of services common to the ISO Common Application Service Elements (CASE) and the SNA LU 6.2 protocol.

3.2 Kernel Support for Modular Protocols

A framework that supports functional layering will obviously support the modular development of protocols within the kernel. Also, as with applications, appropriate service interfaces will enable developers to implement higher layer protocols in a manner that frees them from

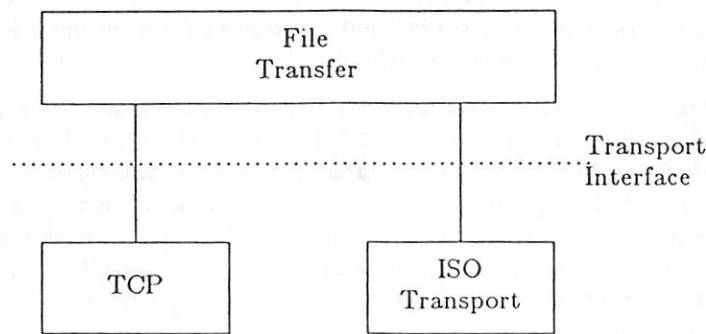


Figure 3. Protocol Independence

knowledge of the underlying protocols or communications media.

The transport interface supports protocol independence for protocols above the transport layer. In addition, a data link layer service interface (based on the IEEE 802.2 Service Definition^[4]) has been defined to hide details of the underlying communications media from network layer protocols. In this way, new communications media can be supported with no change to protocols above the link layer.

Figure 4 demonstrates the ability to develop an internetworking protocol that can support several subnetworks, where each subnetwork is implemented over a different communications medium using the common link layer interface.

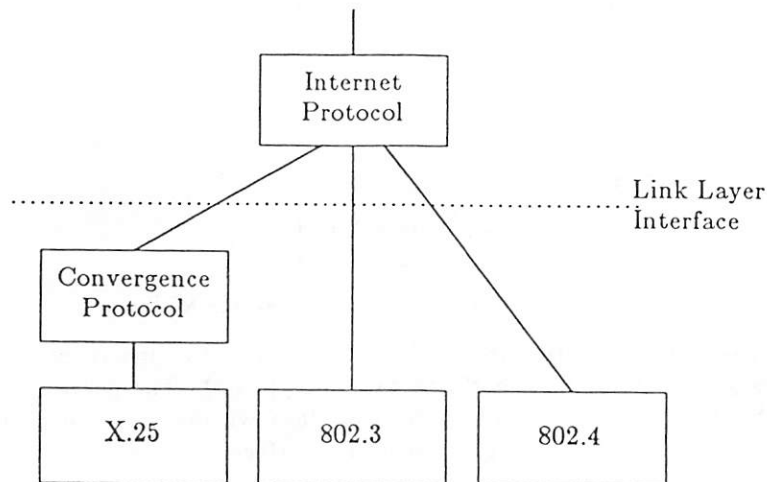


Figure 4. Media Independence

3.3 Support for New Protocol Families

The third goal of the networking framework was to easily support new protocol families under System V. Because the framework service interfaces are based on standard ISO service definitions, they achieve the protocol and media independence that is necessary for folding a large set of protocol families into the system.

4. Kernel-level Service Interfaces

We can now convert the abstract representation of the System V networking framework into the specifics of the implementation. Within the kernel, STREAMS supports the separation of protocol functions into separate modules that can be interconnected in a layered fashion. As such, this mechanism was chosen as the foundation for our networking framework. Information

is passed between STREAMS modules in the form of messages. A service interface, then, can be defined by specifying the content of messages that are passed between modules, the semantics of each message, and the allowable sequence in which the messages may be passed.

Our service interfaces are modeled after the OSI Service Definitions. The primitives used to specify a given service map directly into STREAMS messages. One characteristic of the OSI primitives is that they consist of two components: service parameters and user data. The service parameters define the primitive and any associated information. For example, a CONNECT primitive might include an identification of the primitive and the address of the entity to which a connection is desired. Furthermore, some OSI primitives support the transfer of user data with the primitive. The T-CONNECT-request primitive in the ISO Transport Service Definition is an example.

A new STREAMS message type (M_PROTO) was defined to support the structure of these primitives. This message type (Figure 5) consists of one or more linked message blocks, where the first block contains the service parameters of a primitive and any subsequent blocks contain user data.

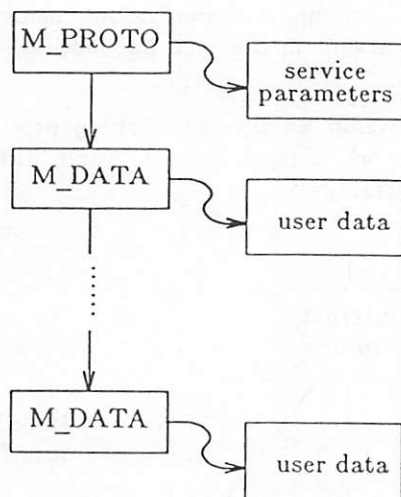


Figure 5. STREAMS Protocol Interface Message

An advantage of this message structure is that it supports the insertion of a filter module between two protocol modules (e.g. to perform data encryption). The message structure enables these filter modules to locate and manipulate user data without requiring knowledge of the format of each particular primitive of a given service interface.

Armed with the new STREAMS message type, we were able to define the transport and link layer interfaces described earlier. The interfaces are defined by specifying the contents of each M_PROTO message that may be passed between the service user and service provider. Furthermore, a state table has been formulated to specify the allowable sequence of primitives (messages) that can be passed through the interface.

4.1 AT&T Transport Interface

Based on the ISO Transport Service Definition, the AT&T Transport Interface was designed to support applications and higher layer protocols in a protocol-independent manner. The interface specifies access to both the virtual circuit and datagram services needed by a large group of applications and protocols. Also, if appropriate, protocol-specific services can be accessed through the interface.

The transport interface can be accessed from either kernel or user level. Kernel protocols and applications access the interface by passing M_PROTO messages to and from a chosen transport

protocol module. User-level access is provided through a new library, which will be described later.

The System V Remote File Sharing service^[5] has been implemented over the transport interface, enabling it to run over a variety of transport protocols. We've demonstrated the ability to execute this service over TCP, the ISO Transport Protocol, and several proprietary transport protocols.

4.2 AT&T Link Layer Interface

A second service interface has been defined to support the services of data link layer protocols. This interface is based on the IEEE 802.2 Service Definition and is intended to support media independence for network layer protocols. Currently, only a datagram service (i.e. LLC Class I) is specified by the interface. The link layer interface is intended to provide transparent access to the 802 family of protocols, as well as the Ethernet** protocol. In addition, a special convergence protocol can be implemented to support access to an X.25 subnetwork below the interface. Unlike the transport interface, user-level applications are not expected to be written directly to the link layer interface. As such, a user library has not been provided.

5. User-level Service Interfaces

STREAMS provides user-level access to kernel-level service interfaces through the UNIX system call interface. User data intended for a STREAMS module is copied into a message and passed downstream. Similarly, messages arriving from downstream modules are copied into user-supplied buffers. Because a user process can send messages to a protocol module and receive messages from a protocol module, it is capable of communicating through a service interface.

Ritchie's STREAMS implementation supported the transfer of data between a user process and a module using the *read* and *write* system calls. These system calls provide a byte stream interface to user processes. To help support the new M_PROTO message structure, we added two new system calls. *putmsg* enables a user to send a message downstream, and *getmsg* supports the inverse operation on incoming messages. Each of these system calls preserves message boundaries and provides separate buffers for service parameters and user data. As such, they simplify the creation and parsing of M_PROTO messages as they're passed between user processes and kernel-resident STREAMS modules.

The service interface between a kernel-resident protocol module and a user process is defined in terms of M_PROTO messages. The system call interface supports the transfer of these messages across the user-kernel boundary. In this way, we have avoided the implementation of any specific service interface (e.g. transport) at the system call level, and instead have provided a simple pass-through capability. A user process can communicate with the service interface of its choice. For example, a user program that requires transport layer services can use the messages defined by the transport interface to interact with a transport protocol module. Furthermore, we expect kernel-resident protocol modules to migrate to peripheral devices, and user-level protocols to migrate into the kernel. As this occurs, the existing STREAMS system call interface is flexible enough to support the new interfaces without change.

Figure 6 illustrates how protocol migration is supported. As the session layer protocol migrates into the kernel, new applications can be implemented to the session layer interface, while existing applications (and the Remote File Sharing service) can continue to communicate directly with the transport protocol using the transport interface. The system call interface requires no change to support both interfaces.

** Ethernet is a trademark of Xerox Corporation

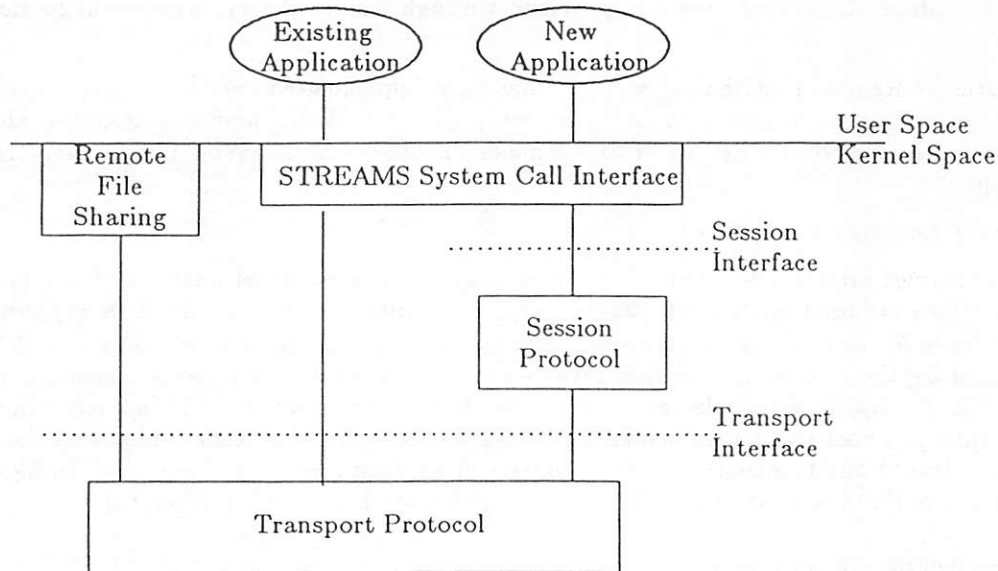


Figure 6. Protocol Migration

5.1 AT&T Transport Interface Library

To simplify the interaction between a user process and a transport protocol, a user-level library has been implemented for the AT&T Transport Interface. This library hides the STREAMS message-based interface from the user, and provides a function call interface for networking applications. As described earlier, the transport level is critical because it provides a common set of service among all major protocol suites. Using the Transport Interface library, networking applications can be implemented to run over any of these protocols. Furthermore, since the transport layer inherently hides media information from its users, applications can enjoy media independence.

uucp and *cu* have been redesigned to operate over the AT&T Transport Interface using this library. In this way, we have supplied protocol independent file transfer and remote terminal services for the UNIX system.

An optional module has also been provided with the transport interface to provide a simple *read/write* interface over an established transport connection. This enables existing user programs (e.g. *cat*) to process data over an established transport connection.

Another set of modules provides a *tty* interface over a transport connection. A network *tty* module is pushed onto a stream above the transport interface to transform a transport connection into a pseudo-*tty* device. A line discipline module sits above the network *tty* module to provide normal *tty* character processing for users. This configuration supports remote terminal services.

6. Future Directions

The framework is in place for providing networking services under the UNIX system. Initially, service interfaces have been defined at the transport and link layers to provide protocol and media independence for a large class of protocols and applications. Services have been developed which substantiate this claim. Work is underway to implement new protocols and applications to these interfaces. We expect to continue our efforts in identifying and defining important service interfaces. One such interface may be a CASE/LU 6.2 interface for network applications. Also, a standard directory service capability is being investigated to simplify the design of future network services.

7. Acknowledgements

The design of the System V networking framework is based on the early STREAMS work of Dennis Ritchie. Many individuals have contributed to the framework design. In particular, Maury Bach, Tom Butler, Her-daw Che, Tom Fritz, Ian Johnstone, and Keith Kelleman contributed to the design and development of the system. Also, we would like to thank Peter Honeyman and members of the Computer Systems Research Department of AT&T Bell Laboratories for their feedback and suggestions during early stages of our design.

REFERENCES

1. D. M. Ritchie, "A Stream Input-Output System," *AT&T Bell Laboratories Technical Journal*, 63:8 (October 1984).
2. CCITT Recommendation X.200, "Reference Model of Open Systems Interconnection for CCITT Applications," (1984).
3. ISO IS 8072, "Information Processing Systems - Open Systems Interconnection - Transport Service Definition," (1984).
4. ANSI/IEEE Std 802.2, *Logical Link Control*, IEEE/Wiley-Interscience (1984).
5. A. P. Rifkin, "RFS Architectural Overview," *USENIX Conference Proceedings*, Atlanta, Georgia (June 1986).

OSI AND TCP/IP PROTOCOLS ON A UNIX SYSTEM V

Jean Marc Fenart

Marc Fievet

Christian Huitema

Bernard Martin

Annie Remille

Guy Vaysseix

GIPSI-SM90¹

c/o INRIA

BP 105

78153 LE CHESNAY CEDEX

FRANCE

---mcvax!inria!gipsy!fenart

Abstract: This paper describes our implementation of OSI and INTERNET protocols in our System V based kernel. As a user/protocol interface, we chose to implement the IPC socket interface. The main problems encountered in the port of the IPC socket and INTERNET protocols and the implementation of the OSI protocols are presented. In this paper, several issues related to multiprotocol systems, protocols implementation and user/protocol interface will be addressed.

1. Introduction

GIPSI-SM90 is a joint research & development group founded by CNET, INRIA and BULL SEMS. Its goal is to make hardware and software developments to bring out a powerful scientific workstation. The SM90 (680xx based) workstation [1] is based upon a multiprocessor architecture developed by CNET (Centre National d'Etudes des Télécommunications).

GIPSI-SM90's R&D areas include:

Hardware (virtual memory management, Lisp and Prolog processors, array processors)

¹ GIPSI-SM90 is sponsored by the French Ministry of Research and Technology under the contracts nos 83-B1032, 84-E0651 and 85-B0524.

Unix² kernel(multiprocessor implementation, disk management)

Graphics (multiwindowing, GKS interface, PostScript³ interpreter)

Obviously such a workstation needs a powerful and flexible access to local and long haul networks and must be able to communicate with various computers. This implies to offer an easy user interface to the mostly used network architectures in the scientific community.

2. Implementing the socket interface on Unix System V

In order to develop network oriented applications on the SM90 workstation, we needed a user/protocol interface. Since the stream interface was not yet available, we chose to implement the IPC socket interface such as in the Berkeley 4.2BSD release [3] for VAX's which offered:

- An asymeric interface based on the client/server model (*connect,accept*).
- A multi scheme of adressing (*domain*).
- A multi protocol architecture (*protosw*).
- A network interface multiplexing (*ifnet*).
- Memory management for network protocols (*mbuf*).

2.1. Hardware dependant routines

The SM90 and VAX hardware architecture are different, so our first work was to deal with the hardware incompatibilities. In this version of SM90, we had no paging and all the memory management routines for the *mbufs* and *clusters* were rewritten. The critical point was to define a reasonable number of *mbufs* and *clusters* which are allocated as static data in the kernel. In the current implementation, we have 128 initial *mbufs* of 128 bytes and 4 *clusters* of 1 Kbytes.

The clock resolution is also different and timeout intervals were revised.

2.2. Incompatibilities between System V and 4.2BSD

If we turn down all the problems involved by procedure name collisions, the major incompatibilities between the two systems are the file structure differences, the *ioctl* and vectored I/O formats.

² Unix is a trademark of ATT Bell Laboratories.

³ PostScript is a trademark of ADOBE Systems

a) File Structure differences

4.2BSD defines a set of generic operations on files *fileops* and for each of them its interpretation depends of the object type (socket or inode) . We introduced a new type in the file structure and programmed an explicit test on the file type (socket or inode). If it is a socket, a 4.2BSD routine is called. A pointer to a socket (usually to an inode) is added in the file structure in order to manage the AF_UNIX domain.

The *file* and *inode* structures were modified and the *SIGIO* and *SIGURG* signals are added.

b) ioctl and vectored I/O formats

At the opposite of System V, 4.2BSD uses a general format for the ioctl requests which is based on the triplet <type, size, value> . The *uio* structure is introduced like in 4.2BSD in order to implement the vectored I/O routines (*sendmsg/recvmmsg*) which allow the fragmentation of the I/O buffers. An *iovec* vector which is referenced by the *uio* structure associated with the current I/O operation gives an access to the different fragments of the I/O buffer.

In order to avoid modifications in the 4.2BSD routines, these new structures have been implemented for the sockets operations, but are not used by the other I/O operations.

c) Domains

In addition to the AF_INET and AF_ISO domains, the AF_UNIX domain was implemented. But since the named pipe (FIFO) mechanism cannot be easily mapped onto the AF_UNIX local domain, we chose to keep the standard System V implementation of pipes. The main interest of the AF_UNIX is for testing the socket implementation in the local environment.

2.3. 4.2BSD systems calls supported

We added 19 new system calls for the socket interface in the kernel:

sethostname(), *gethostname()*, *socket()*, *bind()*, *listen()*, *accept()*, *connect()*,
socketpair(), *sendto()*, *send()*, *recvfrom()*, *recv()*, *sendmsg()*, *recvmmsg()*,
shutdown(), *setsockopt()*, *getsockopt()*, *getsockname()*, *getpeername()*.

and 4 system calls used by various networks applications:

sethostid(), *gethostid()*, *setreuid()* and *setregid()*.

The size of the code for the socket implementation in the kernel is 28 Kbytes.

3. TCP/IP protocols and applications

INTERNET (IP, TCP, UDP) protocols are widely used in the scientific community and IP/UDP offers a nice and efficient datagram service. We thus implemented:

the 4.2BSD INTERNET protocols suite including ARP, the Trailer option, and checksum control,
most 4.2BSD utilities: *ifconfig*, *netstat*, *ftp*, *ftpd*, *rlogin*, *rlogind*, *sendmail(SMTP)*, *rsh*, *rshd*, *rwho* and *rwhod*.
and the 4.2BSD libc extension relevant to networking applications.

3.1. Driver Interface

The Ethernet⁴ driver is composed of a physical driver and a logical driver. The physical driver must support various kinds of protocols (X25, INTERNET, remote disk) and thus must be able to distinguish between IEEE802.3 and Xerox formats for Ethernet frames. It must also offer a simple interface to the logical drivers. The link between the INTERNET protocols is done by a logical driver which is a simplified version of the 4.2BSD Interlan driver. With its introduction, no modification was required for the INTERNET protocols implementation (net and netinet modules).

The *if_rubaget* and *if_wubaput* routines which move data from an Ethernet controller buffer to a chain of *mbufs* or in the reverse order, have to be rewritten in order to take into account the no paging environment.

3.2. INTERNET protocols

The basic interface between the Ethernet driver and IP routines in 4.2BSD is an input and output queue. When an Ethernet frame is received, it is copied into *mbufs* which are linked in an input queue, a software interrupt is notified and a return from interrupt is performed. When the software interrupt is taken into account, an *ipintr* (ip interrupt) is called and the *mbufs* linked in the input queue are analysed. This software mechanism allows to run the interactions with the Ethernet controller at a high priority and network protocols at a low priority. In our system, frames are buffered in the Ethernet controller, so we did not need to implement the software interrupt mechanism which was moreover too complex in a multiprocessor environment.

⁴ Ethernet is a trademark of Rank Xerox Coporation

We discovered some bugs in the 4.2 BSD INTERNET protocol implementation that were hidden by the virtual memory mechanism of the VAX. In a no paging environment the memory is not illimited and most of errors were induced by a shortage of *mbufs* which were sometimes not deallocated and some other time too many times deallocated. So we have implemented a control algorithm on the mbuf allocation. The time slices for the timers have been modified, some of them beeing too large on the VAX version (*FIN_WAIT* for exemple) and requiring too much ressources. The computation of time is performed on integer values in order to avoid the floating point operations. We reduced some data structures, *arptab* or *tcp_ndebug*, too generously allocated.

3.3. Utilities

For the utilities we have solved the classical problems of program transportation from a VAX with paging to an architecture based on MC680xx without paging: byte order, null address, stack overflow,... Some differences between System V and 4.2BSD are confusing:

- some usual structures are different,
- *free* and *malloc* do not run in the same way and some signals (*SIGCHILD* for exemple) have not the same use.

In our system, we have not implemented the *select* system call and it is replaced in some utilities like *rlogin* by a cooking of non blocking *read* and *fork* syscalls.

The *setuid* and *setruid* system calls were implemented for the programs which perform many *set user id "root"* (*bind* on privileged ports).

All the 4.2BSD extensions are gathered in the libinet library:

closedir, *dirbsd*, *gethostbyadr*, *gethostbyname*, *gethostent*, *getnetbyaddr*, *getnetbyname*, *getservbyname*, *getservbyport*, *getservent*, *opendir*, *rcmd*, *readdir*, *rexec*, *ruserpass*, *setuid*, *setruid*...

For the *ftp* and *ftpd* utilities, we simulated some 4.2BSD system calls, *rmdir*, *rename* and *mkdir*, which does not exist in our version. On the other hand, we have implemented the pseudo tty for *rlogind* and the bitmap management.

In conclusion, we are impressed by the very good portability of the INTERNET protocols in an environment which was so different from 4.2BSD: different computer (16 bits), multiprocessor environment and SystemV operating system. The machine dependant programs which have to be changed are the Ethernet driver, the memory management and the checksum routine.

The size of the code in the kernel for the INTERNET suite is 35 Kbytes. The throughput measured between the SM90 workstation and different machines (VAX,SUN,RIDGE) ranges from 20 kbytes/sec to 50 kbytes/sec.

4. Why the OSI protocols

Although the actual TCP-IP suite gives access to a large number of machines, we also felt the need for the OSI protocols. Users want to use various applications which are already normalized or being normalized by ISO or CCITT, like the X400 electronic mail, the FTAM file servers or the X.DS directory services.

Also, the OSI architecture is interesting when using the public networks, a mandatory requirement in Europe. It is indeed feasible to use IP over X25, but the OSI connection oriented transport service will make a much more efficient use of the X25 or X21 packet-switched or circuit-switched networks.

The OSI 7 layers architecture offers the opportunity to choose between four "transport classes" (CCITT Recommendations X214 and X215), and to negotiate the "session service elements" (CCITT Recommendations X224 and X225). Implementing all OSI protocols variants implies a huge effort, so we had to choose first a subset of these possibilities. Since as a first application we wanted to run on the network the X400 message handling service, for which the class 0 of the transport protocol is mandatory, we decided to first concentrate our effort on this simple class.

Similarly, as X400 requires the fairly complete "Basic Activity Subset" of the session service, we decided to implement the session in a flexible way, to provide the functionalities of the "BCS", "BAS" and "BSS" profiles.

The class 0 transport protocol is run on top of the connection oriented network service furnished by X25, as defined in ISO 8878. The X25 virtual circuits can be provided over point to point connections, or on Ethernet as defined in ISO 8881. This choice of implementing class 0 on top of X25 rather than the NBS favoured class 4 over ISO-IP solution was justified by two reasons:

- the X25 code was already available. It was first written to offer an X25/X29 service through the tty interface, both on Transpac and Ethernet.
- this X25 layer can be run over point to point connections and over Ethernet and provides an X25 switched service.

Up to now, we have not felt the need for more sophisticated transport classes. The quality of service of X25 connections is sufficient, even in an international environment. Moreover, it is always possible under the CCITT defined rules to "negotiate down" to class 0 if the remote caller proposes any other class on a virtual circuit. However, the development of the class 4 protocol is now underway; the choice of protocol classes will not impact the session service visible by the user.

In order to have a good throughput on the network, a large part of the code had to be implemented in the Unix kernel. We took advantage of the availability of the "socket" environment to implement the transport protocol and the basic functionalities of the session protocol as "protocol layers" in the "ISO domain". However, this environment was initially tailored to the need of the TCP-IP protocols and we had to slightly modify it.

5. Implementing OSI protocols under the socket interface

5.1. User/socket interface issues

The transport and session services are connection oriented, packet oriented and reliable. So it might seem obvious to use the `SOCK_SEQPACKET` type of interface. But from [2] we discovered that this approach cannot be followed without change in the socket code. Moreover, it would have let us to use the `PRU_ATOMIC` flag, with the side-effect of limiting the length of the data that a user could send in a single system call. This is incompatible with the specification of the session service, where the size of messages is only limited by an end to end negotiation at the application level. The actual implementations of the ISO applications may map on a single "session service data unit" such objects as an image, a facsimile page, or a whole document sent as an X400 message.

Thus, we decided to use the `SOCK_STREAM` type of interface and to modify slightly the socket code in order to maintain message boundaries. So ISO connections can be used as byte streams, but the "atomicity" of the *send* operations will be preserved. The users will send a certain amount of data, which will be transmitted as a sequence of packets (TPDU) using the transport protocol segmentation facility. The receiving application will give in the *recv* operation the maximum number of bytes that it is ready to accept. Then the system will fill up this BUFFER up to the maximum length given by the user or the end of the sequence of packets. An *ioctl* command can be used to check whether the packet boundary was reached during the last *recv*.

Thus, we only had to add a few lines to the code of the *send* and *receive* syscalls.

5.2. The session service interface

Apart from the delimitation of packets, the OSI transport and the TCP-IP interfaces are almost the same. The only difference resides in the possibility to carry a limited amount of Transport user data in the connection request queries. If we had to offer this service to the transport user, we would have to deal with the limited semantics of the *connect* and *accept* primitives. To solve this problem, we envisioned first to implement a "user data" parameter in *connect* and *accept*, or to add a *confirm* primitive. But further analysis showed that this transport connection data are never used by the session protocol. So we decided to keep the actual syntax of the socket system calls.

But the same could in no way be true for the session service. User data are used in the connection request and the connection response, e.g. to carry passwords or to perform end to end negotiations. These primitives also carry a large number of parameters, which would be uneasy to map on a few interface toggles. Moreover, the user can request, in the course of the connection, such services as starting an activity or inserting a checkpoint, hereagain filling several parameters fields of the session messages.

Implementing these services in the kernel would have necessitated complex *ioctl* calls, or an extension of the "socket" primitives, e.g. to include *check__point(socket)* or *give__turn(socket)*. On the other hand, implementing the session protocols as users level subroutines on top of the transport service would have resulted in something very inefficient, not to mention the impossibility to let the session connection "look like an Unix file". Thus, we had to implement a tradeoff:

For a session connection, the *connect* and *accept* commands will correspond to the establishment of the underlying transport connection. The *bind* command will indeed be used to specify a transport address.

Two data flows will be made available for the user, one for normal data, corresponding to "session service data units", the other for control data (out of band).

The "complex" session primitives will be processed by user-level subroutines that will format session protocol data units and send them on the "control flow". An application waiting for normal data will get an *EOOBIN* error if out of band data arrives, and can then use the provided subroutines to analyse the event.

We are still working on a new definition of the session implementation, where the user will be able to request a "profile" in order to request automatic processing of some session procedures, like the handling of the turn ("data token") or that of "minor checkpoints".

5.3. Protocol/socket interface issues

To maintain messages boundaries, we needed a mechanism to prevent the users from sending data until the last part of the previous message has been sent. The control flow strategy built in the socket interface based on the high water mark on the send queue does not provide for such a mechanism. Thus we changed slightly the socket code. One may argue that this difficulty comes from our use of the *SOCK_STREAM* interface as a *SOCK_SEQPACKET* one. Nevertheless, the socket control flow mechanism is not as general as it could be.

The socket interface is making implicit and explicit assumptions on the behaviour of the underlying protocol. One obvious example is that the two syscalls *listen* and *accept* are too many to be mapped on the ISO Connect Confirm primitive and that as discussed previously the very limited number of arguments given through them forbids any meaningful use of those two syscalls by the ISO protocols.

Another example can be given by the way the *sonewconn* is implemented. This function should be used to notify a "server" socket of an incoming call. But this function always allocate ressources for the new connection by calling the *PRU_ATTACH* procedure. This means implicitly that no ressource has been yet allocated. But unfortunately this is wrong when the lower layer protocol is connection oriented - i.e. X25. Indeed the Transport must first confirm the X25 connection before it can know the "name" of the receiving socket that is given in the Transport Connect Protocol Unit. Thus it must first allocate a Protocol Control Block for the new incoming connection and later deallocate the PCB handed by the *sonewconn* function.

As a conclusion we think that many functions named *soxxxxxxx* in the socket code should be called *tcpxxxxxxx* or *ipxxxxxxx* and that the socket interface should do less in order to be more general.

5.4. Ifnet interface issues

When we started this work, X25 was already implemented to provide a X25/X29 service. So this code did not use the *mbuf* memory management routines. Thus using the ifnet interface between X25 and the drivers - e.g. Ethernet - would have meant a lot of work and/or poor performances.

The use of ifnet to interface with the X25 service raises three main problems:

- memory management for the PAD,
- mapping of the connection oriented X25 service with the ifnet interface,
- our X25 software must be able to use different drivers and hardware devices.

Those problems might be solved by rewriting part of our X25 code and also part of the "ifnet" code and part of the routing procedures (*rtalloc*, *rtfree*). Since we wanted to use the ISO services as soon as possible, we decided not to use the ifnet interface but our much simpler Transport/X25 interface.

5.5. Results

Even with the few problems mentioned above, the socket interface allows us to implement the Session and Transport protocols rather quickly - i.e. 6 months.

The X400 electronic mail is currently running and using the session service to deliver messages either on the local network or through the public networks (TRANSPAC, DATEX-P, PSS) to foreign research institutes which are using a compatible software (EAN). A ISO remote shell has been also implemented.

About twenty lines of code were added to the socket interface software. The size of the code added to the kernel are 7 kbytes for X25 over Ethernet, 6 kbytes for the Transport and 6 kbytes for the Session. The average throughput at the session level over Ethernet is 30 kbytes/sec.

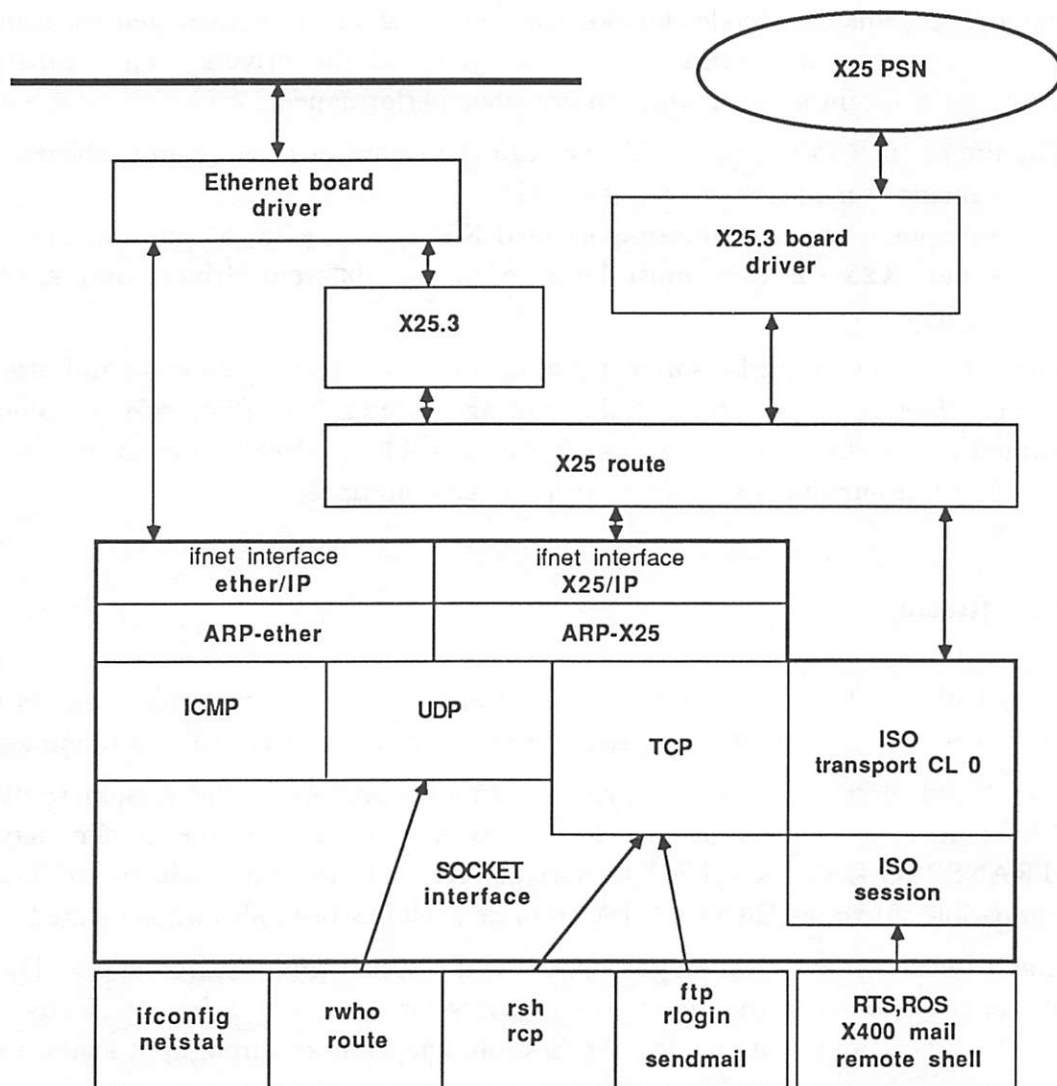


Figure 1: network architecture

6. Conclusion

With a reasonable amount of effort we succeeded in implementing a multiprotocol system where System V, the socket interface, the OSI and TCP/IP suites cooperate nicely sharing resources and various device drivers.

This system is currently running on a lot of SM90 workstations connected on the INRIA local network together with 3 VAX's and a few SUN and APOLLO workstations. Various applications such as remote login, file transfer and electronic mail are used daily on this network.

The following developments are underway:

- Implementation of the select system call,
- Implementing a Transport class 4 over Ethernet,
- Remote disk over UDP,
- Full Network File System (based on Sun Microsystems product),
- Development of ISO standard applications: directory services, FTAM.

7. Bibliography

- 1 - Abramatic J.F.; "The SM90 Workstation" Proceedings of the First IEEE Conference on Computer Worstation, San Jose Nov 1985.
- 2 - O'Toole, J; Torek, C. and Weiser, M.; "Implementing XNS Protocols for 4.2bsd" USENIX Winter Conference Proceedings, Dallas, 1985.
- 3 - Leffler S.J.; Joy W.N. and Fabry R.S.; "4.2bsd Networking Implementation Notes" University of California, Berkeley, Report # UCB/CSD 83/146.

A MULTIUSER MULTIPROCESSOR
BENCHMARK
TO COMPARE UNIX SYSTEMS

Philip M. Mills

NCR Corporation
3325 Platt Springs Road
West Columbia, S.C. 29169

Introduction

Despite all the evaluation of Unix systems using benchmarks, no readily available benchmark today provides an accurate estimate of a multiuser Unix System's overall processing effectiveness. Published articles showing benchmark results as well as the results from executing benchmarks, typically provide tables of data containing measurements of different parts of the system, leaving one with the job of interpreting the results as best they can. What one really wants to know is will a set of application programs execute faster on one Unix system compared to another Unix system.

The problem with the results provided by most benchmarks is that one cannot accurately infer which system can execute a set of application programs the best for the following reason:

- (1) The benchmark does not contain a workload that provides an adequate simulation of a Unix multiuser application environment.
- (2) The benchmark ignores the system's ability to overlap operations.
- (3) One is unable to correlate the time to perform simple operations with the system's overall ability to process work.
- (4) The benchmark cannot evaluate Unix systems with multiple main processors.

To overcome the difficulties just described, a self-measuring portable multiuser multiprocessor Unix benchmark has been developed. This benchmark provides:

- (1) A reasonable approximation to a multiuser multiprocessor Unix application environment.
- (2) A single measure of the overall system's processing effectiveness.
- (3) An easy way to compare the relative overall system's processing effectiveness between systems.
- (4) A method of relating application requirements to the benchmark results.

In the development of the benchmark the first step was to establish the benchmark objectives. The overall objective was to compare the performance of different Unix systems or different configurations of the same Unix system. The factors to consider in the benchmark development were:

- (1) The basic functionality of the system hardware and how it works.
- (2) The major elements of user application process executing in a Unix based computer system.
- (3) Performance measures.

In other words if the appropriate elements of the Unix application process execution were included in the benchmark and they exercised the hardware functionally as it would be in real use and a performance measure provided an accurate estimate of the system's overall processing effectiveness then the initial objective could be met.

In the sections to follow system functionality, the elements of Unix application process execution, and performance measures are discussed, followed by a description of the benchmark developed.

Functionality

A Unix computer system has three major areas of functionality that a workload should exercise at the same time. The first is the processing by one or more central processing units (CPU) where all the calculations are performed; the second is the disk subsystem where the computer stores data files; the third is serial I/O subsystem which controls such items as video displays, terminals and printers.

In a multiuser Unix system there are many user processes or programs executing at the same time. This type of processing allows the I/O processing of some user programs to be overlapped with the CPU processing of other user programs. The result of overlapping the I/O operations with CPU operations depends on the amount of intelligence built into the I/O processors, the level of multiprocessing and the ratios of CPU processing time, disk subsystem processing time and TTY subsystem processing time.

An intelligent serial I/O subsystem may handle interrupts from multiple on-line users, provide a slave serial I/O processor, handle buffering for both input and output and may even execute parts of the Unix kernel's TTY code. Besides providing overlapped serial I/O operations for the terminals and printers with the main CPU processing, the intelligent serial I/O subsystem may also greatly reduce the amount of processing required by the main CPU. The reason for this reduction is that functions or processing that used to be done by the main CPU are now done by the serial I/O subsystem.

The serial I/O subsystem often consists of more than one TTY controller that handles typically 8 or 16 TTY lines or ports.

The TTY controllers vary in capability from system to system but usually the TTY controllers can overlap their operations with each other. This affects the total number of characters the TTY subsystem can transmit and receive across all the lines.

An intelligent disk subsystem will have one or more disk controllers and disk drives and may also have a slave processor. The disk subsystem handles the control of the disk drive operations, handles interrupts, handles buffering and provides a high speed DMA data transfer in and out of main memory. If a slave processor is available it may be used to execute some of the Unix kernel file code. Intelligent disk subsystems provide overlapped operations with the main CPU and also reduces the load on the main processor. In some disk subsystems, operations on separate disk drives can also be overlapped with each other producing a much higher effective transfer rate for the disk subsystem.

In the multiuser execution of the Unix application processes the operations performed by the CPU subsystem, the disk subsystem and the TTY subsystem are overlapped in time. Also within each subsystem many elements may operate in parallel or overlapped, such as multiple CPU's, multiple disk controllers, multiple disk drives, multiple TTY controllers and special I/O processors. Even in a small PC the floppy disk controller operation can be overlapped with the main CPU execution.

The Workload

A sound approach to evaluating the performance of various computer systems is to take the application or sets of application programs that make up the user environment and execute them on each computer system to be evaluated. It is most important that the set of application programs be truly representative of the overall user environment. This approach is often not feasible because the set of applications do not exist or it would take too much manpower to port the application programs to run on each computer system being evaluated. Also it may take too much time to execute a large set of applications on a number of computer systems.

For the reasons just given portable benchmarks that are easy to obtain and execute in a relatively short period of time are often used to simulate the application program environment. The workload for a benchmark should contain the following Unix user application process elements.

- (1) A broad weighted mix of basic CPU operations used with different data referencing modes.
- (2) A number of user processes executing in a multiprocess mode.
- (3) The terminal I/O taking place with a number of terminals at the same time across each of the TTY controllers.
- (4) The disk I/O taking place over the different disk controllers and disk drives with a number of files scattered over each

disk drive surface to cause head movement.

- (5) The workload should provide for simultaneous execution of the elements described in (1) through (4). This is handled in part by the nature of the Unix multiuser environment.
- (6) The workload should be setable or configurable to different Unix system configurations.

Another issue of critical importance to the workload is the mix and ratio of the CPU, disk and TTY operations to use in the workload. This issue will be addressed in the actual description of the benchmark developed.

A point worth repeating is that the objective of the workload is to provide a reasonable simulation of a Unix multiuser application environment.

Performance Measures

For discussion sake, performance measures can be divided into four levels on a Unix system as follows:

- 1) The overall system level
- 2) The subsystem level
- 3) The system call level
- 4) The hardware level

The hardware level is the lowest level and would consist of such measures as bus speeds, memory reference time, time to do an integer add, etc. The system call level would measure the time to do forks, getpid, sbreak, etc. The subsystem level would consist of measures of the capacity of the CPU subsystem, (one or more CPU's), the TTY subsystem and the disk subsystem.

The capacity of the CPU subsystem would be the number of units of code with a broad weighted mix of basic CPU operations used with different data referencing modes that can be executed in a fixed amount of time. The TTY subsystem capacity would be the total number of characters that may be transmitted and received in a unit of time; for example, a second. The disk subsystem capacity would be the total number of disk blocks or bytes that can be handled in a second. A secondary objective of providing CPU, TTY and disk subsystem capacity information was established for the benchmark.

At the overall system level there are two main results that are important to measure in a Unix based computer system: responsiveness and throughput. Responsiveness measures the system's ability to respond to a work request in a given amount of time. Throughput measures the total amount of work performed over a given period of time.

The overall system level measures of response time or throughput are the only accurate estimates of the system's processing effectiveness. The problem with trying to make an inference as to the system's pro-

cessing effectiveness from lower level measurements (Levels 2, 3, and 4) is that the effect of overlapping operations cannot be estimated and it is not known how to correlate simple operations with the system's overall ability to do work.

The measurements made at levels 2, 3, and 4 do provide general insight into why the system performs the way it does and the measurements do provide information that is quite useful in tuning Unix systems. This information can best be used (and is) by organizations marketing Unix computer systems who are able to modify the hardware and Unix software to improve performance.

The point to be made here is that if a Unix system is being evaluated for a purchase the overall system level measures of responsiveness or throughput should be used for comparison in that the system itself will automatically provide the effects of overlapped operations and the speed of low level operations.

Throughput vs. Responsiveness

In the development of the benchmark a decision had to be made to measure throughput or responsiveness. Although response time is a very important performance measure in an interactive Unix multiuser system it requires the use of a remote terminal emulator computer to provide TTY inputs to obtain repeatable results.

Because responsiveness and throughput are both very much related to a computer system's ability to process work the relationship between the two was investigated. The results of this investigation showed that a 20% increase in throughput would provide a 20% or greater improvement in response time. How much greater than 20% the improvement in response time would be depends on the ratio of the time to process an interactive user request to the average user think time. If the think time goes to zero, throughput and responsiveness vary the same from system to system for a given workload.

If x represents the average processing time to execute a workload process containing some CPU, TTY and disk processing, then with n active process in the system the average time from start to finish (response time) non-interactive (no think time) is $n \cdot x$. In an interactive system with think time, response time R equals $Q \cdot X$

where Q is from 1 to n in value
and $Q = n / (1 - P_0) - T/X$

P_0 is the system idle time created by the users who are not supplying enough work to keep the system totally busy and T is the average user think time and R is the average response time. As T goes to zero it makes P_0 go to zero and Q equals n .

The reason for showing the formulas is they show that response time is proportional to x and so is throughput, and that throughput is the special case of response time where the system is not idle waiting on user to supply work.

The conclusion is that throughput is really the best measure of the computer system's ability to process work because it is measured at the system's maximum capacity to do work. For this reason the benchmark measures throughput as a result of executing a particular workload.

The System Characterization Benchmark

The purpose of the benchmark is to compare Unix systems on their ability to do work (throughput) using a simulated Unix multiuser environment with multiple TTY lines, multiple disk drives and one or more main processors. For a fair comparison each computer system should execute the same workload with the same number of disk drives.

The application requirements for a computer system will vary in each environment such as word processing, accounting, graphics, data base management, spread sheet, compilers and scientific computations. For this reason the benchmark (SCB) executes a number of different workloads, each with a different mix of I/O operations. Because of the many different workloads used by the benchmark, it is more of a system characterization than a single benchmark. For this reason the benchmark is called the System Characterization Benchmark (SCB).

The SCB has the following characteristics:

- (1) The SCB measures the rate (throughput) at which fixed units of work can be executed in a unit of time (minutes).
- (2) The SCB takes into account the effects of all overlapped operations executed in parallel, including multiple main processor units, if they exist.
- (3) The SCB simulates a multiuser Unix application environment with
 - a. A broad mix of main processor operations and data references
 - b. Multiple TTY lines with terminals
 - c. Multiple disk drives each with its own set of data files
- (4) The SCB consists of 50 different runs to provide a system characterization that will cover the many different application environments.
- (5) The SCB provides graphical output for a quick analysis and comparisons on the basis of throughput.
- (6) The SCB provides information on the effective rates of the main central processing unit (CPU), the disk subsystem and the TTY subsystem.

The first part of the SCB consists of executing three programs; crun, drun and trun to produce the report shown in Figure 1. that shows the effective capabilities of the one or more main CPU, the disk subsystem and the TTY subsystem respectively.

The purpose of crun is to obtain the relative processing power of the one or more main CPU's. The crun program also provides for multiprocessor systems an estimate of the ratio of the multiprocessor's throughput to a single processor's throughput. If there is only one CPU this ratio is one.

The crun code consists of a weighted mix of basic operations on various types of data references. The frequency of execution of the basic operations involving the different types of data references are weighted according to an internal unpublished compiler study of Unix based C code. Detail information on the mix of operations and data references is contained in the SCB documentation. The time to execute the crun code is measured and used to determine the user CPU time per work unit process.

In setting up the SCB each disk drive specified by the user has 200 files of 20K bytes each placed on it. In the execution of drun the user may specify the percent of files to be read and written on each disk drive. The disk program drun produces the information on the disk subsystem in the subsystem report by reading and writing file blocks randomly over the different disk drive surfaces specified.

The serial I/O subsystem is usually made up of one or more TTY controllers where each TTY controller handles a fixed number of TTY lines/terminals. The TTY controller typically can handle some maximum level of characters per second for TTY output and input across all lines. The information in the subsystem report is obtained by writing 32 character lines to all TTY ports as fast as the TTY controllers can handle them.

The second part of the SCB consists of 50 separate runs controlled by a shell script. Each run is made up of 200 work units. The user processes execute 8 or some setable multiuser level at a time. Each work unit consists of the following components.

- (1) The execution of a copy of xrun code with a CPU user time proportional to the system's crun time.
- (2) A given number of 1K disk block reads and writes.
- (3) A given number of 32 character line writes to a TTY line/terminal.

Each work unit process in the run is identical. The disk and TTY requests are equally spaced in the CPU user time to provide more consistent and repeatable results. Each work unit has its own unique disk read file and write file. The disk files are preassigned to each work unit at random. A work unit process has a unique TTY line during its

SYSTEM UNDER TEST MACHINE A

CPU SUBSYSTEM

Number of Main CPU Processors 1
Throughput for All Main Processors 182.788 CPU only Work Units per Minute
Multiprocessor Vs. Singleprocessor Throughput Ratio 1.000
User CPU Time per Work Unit 0.317 Seconds

DISK SUBSYSTEM

Effective Disk System rate 60.7339 K bytes per Second
Average Time per Disk Request 0.01647 Seconds
Total Disk Requests 7200 Request Size 1 K bytes
Time for All Disk Requests 118.550 Seconds
Number of Disk Units 1
Disk Unit 1 Percent 100.000
Disk Requests per Work Unit 36
Work Units per Minute 101.223

TTY SUBSYSTEM

Total Characters Written per Second 1709.771
Characters Written per Second per Line 213.721
Total Characters Written 768000 request size 32
Total Time To Write Characters 449.183 Seconds
Number of TTY Controllers 1
Number of TTY Lines per Controller 8
Number of TTY Lines Used in Run 8
Work Units per Minute 26.715

Figure 1. The SCB Subsystem Report

execution. On completion the TTY line is given to the next work unit process started.

The 50 runs made by a shell script uses all the combinations of the following disk and TTY levels:

disk blocks 0, 4, 8, 12, 16, 20, 24, 28, 32, 36

TTY lines 0, 30, 60, 90, 120

The disk operations are divided evenly between reads and writes of existing files. The work units executed per minute are calculated for each of the 50 runs and the results are presented in a graphical form.

The Results Of The SCB Benchmark

The results of executing the SCB are presented in the graphical form shown in Figure 2. The left axis represents the number of work units that can be executed per minute (throughput). The bottom axis is the number of disk requests in a unit of work process.

The letters on the graph (A B C E F) represent the different levels of TTY line writes contained in each work unit process.

<u>Letter</u>	<u>Number of TTY line writes</u>
A	0
B	30
C	60
D	90
E	120

The graphical results are produced on a 130 column printer and the border on the graphical report is 8 1/2" x 11" so that it may be placed into reports when desired.

There are 50 points on a graph each specified by a letter. Each point represents the throughput rate in work units per minute. Each point is determined by executing 200 work unit processes each with a unit of CPU user processing and the number of disk and TTY requests specified on the graph.

The SCB results show how a computer system performs under a variety of loads. On moving to the right on the graph, the effect of executing more disk requests is indicated by the steepness of the curve towards the bottom axis. The more limited the disk subsystem in capability the steeper the downward slope to the right. The different letters show the effect of increasing the TTY load per work unit process. The more limited the TTY subsystem in capability the greater the downward distance between the curves of the same letter. Point A next to the left axis shows the system's ability to execute CPU user processing with no I/O.

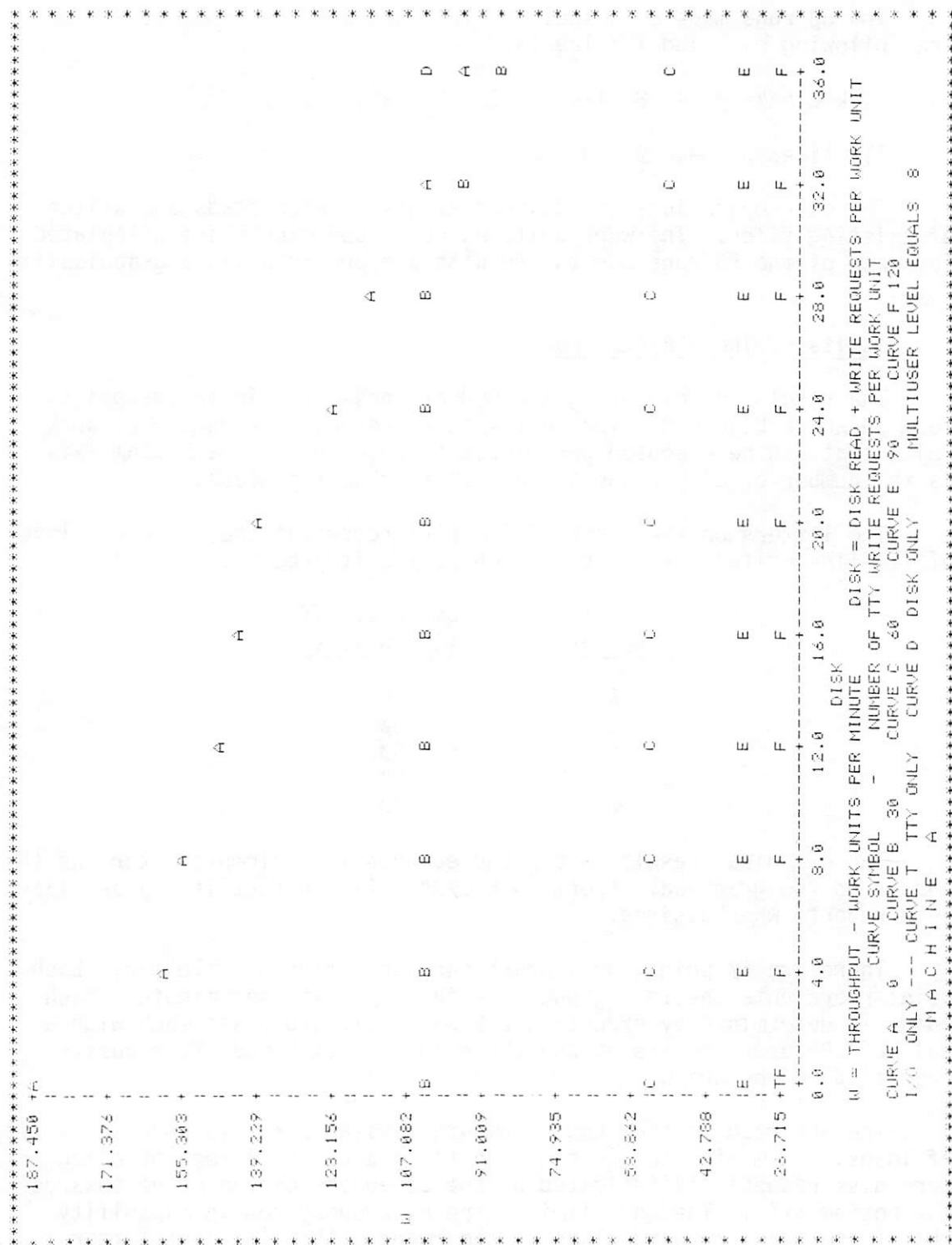


Figure 2 - The SCB Throughput Graph

Facilities are also provided for comparing two different computer systems by taking the ratio of the throughputs of all 50 points and graphing the results shown in Figure 3. The left axis shows the ratio and the bottom axis and the letters (A B C E F) are the same as Figure 2. This graph shows in which area or areas (main CPU, TTY, disk) one system is superior to the other. Ratios greater than one show an increase in capability in percent above one and ratios less than one show less capability in percent under one.

The ratio graph of Figure 3 shows machine A is superior in handling disk requests. Machine B has a CPU/compiler speed that is about five percent faster. The two machines are equivalent in handling TTY write requests.

Relating User Application To The SCB

Using the ratio graph system A can now be compared with system B across a broad mix of different workloads. However, system A may be better than system B (ratio greater than one) over certain regions of the graph and worse than system B (ratio less than one) on other regions of the graph. For this reason users may wish to identify the region of the SCB graph for their typical applications.

To obtain information on an application process or a set of applications, execute the application or a shell script with the sysdata program contained in the SCB software. The sysdata program reads the sysinfo structure before and after the run and computes the ratio of disk activity to CPU time and the ratio of TTY activity to CPU time. The output from sysdata provides a location on both the SCB throughput graph and on the SCB ratio graph.

Having related the application to a location on the SCB graphs the relative performance for the application can now be estimated for every system on which the SCB was run. The run time for the application on a new system can be estimated by multiplying the run time on the old system by the ratio on the SCB ratio graph at the application point.

The sysdata program may also be used to execute shell commands which means whole sets of applications may be related to locations on the SCB graphs. Using the SCB ratio graphs the percent improvements in throughput can be estimated for a number of new systems for different types of applications. These estimates can be obtained without the need to port any of the application programs.

Obtaining The SCB

The software for the system characterization benchmark (SCB) may be obtained free on IBM PC or NCR Tower compatible floppy from NCR by writing on company letterhead to:

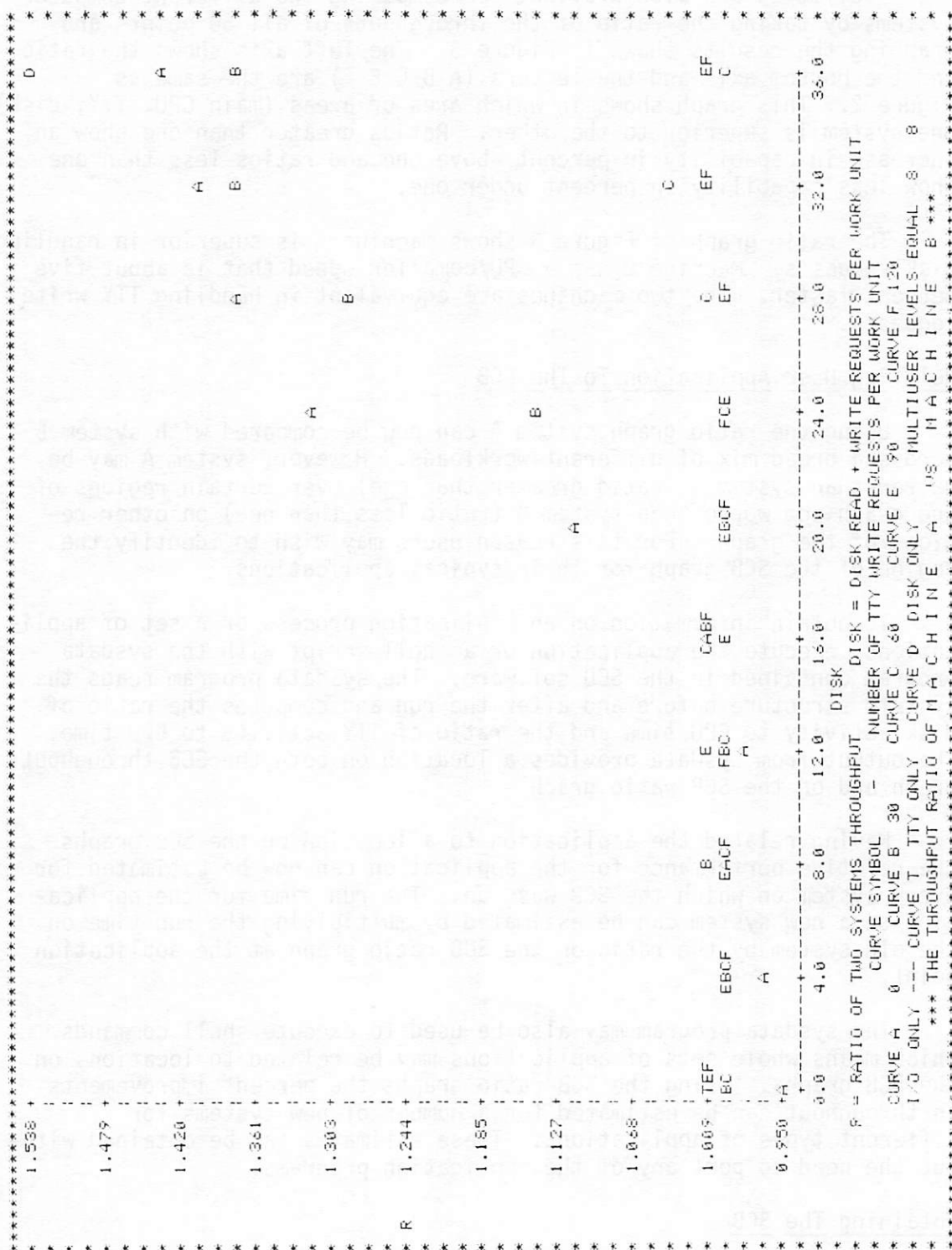


Figure 3 - The SCB Ratio Graph

SCB Distribution
NCR Corporation
3325 Platt Springs Road
West Columbia, S. C. 29169
Attention: Ms. Darlene Amick

The SCB software is also available on the usenet network in net.source and if not posted send mail on the network to usenet address ncrcae:duncan to request reposting of the SCB software.

Conclusions

By the very nature of benchmarks they are an approximation of the user application environment. While there are many different benchmarks measuring various elements of Unix computer systems, a decision to buy a certain computer should be based on the system's ability to handle the overall workload. The time to perform an integer add or read a single disk file is useful information but is certainly not sufficient information to base a computer purchase on.

The problem with gathering a lot of details on Unix computer systems in the form of specifications and the results of making low level measurements is that it is extremely difficult, if not impossible, because of the system's internal complexity to accurately relate this information to a single measure of processing effectiveness. The SCB uses the system itself to relate all the various operations and provides one measure of processing effectiveness. Because the processing effectiveness varies with the nature of the workload the SCB estimates performance for a number of different workloads. Providing workloads with various I/O levels is an important part of the SCB design because many Unix systems are I/O intensive. For a number of applications I/O turns out to be the limiting factor that determines throughput.

The SCB provides a reasonable approximation to a multiuser Unix application environment, provides an easy way to compare the relative processing effectiveness between systems and provides a method of narrowing performance estimates to specific or sets of application programs.

The SCB is easy to set up and run and it provides a comprehensive and reliable benchmark for comparing Unix systems.

Acknowledgment

I wish to thank Myron Merritt for his many hours of discussion and his ideas used in the development of the system characterization benchmark.

A System Call Tracer for UNIX[‡]

R. Rodriguez
decvax!rr

ULTRIX[†] Engineering
Advanced Development Group
Digital Equipment Corporation
Continental Blvd.
Merrimack, New Hampshire 03054

1. Introduction

The performance of the UNIX operating system and its associated tools has long been a fertile ground for interesting work. Every operating system known to this author (including UNIX) allows users to profile their user level code. Profiling a program yields some sort of list or tree representing each function in the program. This list usually contains entries like the number of times each routine was called, or the calling sequence with caller and called information, or the total time spent in each function.

Most of the profile information is oriented toward the local user functions and not to the calls into the system. This is not criticism since the local user functions is what the user can change. However, it does point out an area where a lack of knowledge exists. It is often assumed that system calls are negligible (or something not to be worried about). An over abundant use of system calls in many instances, can result in havoc just because the programmer didn't know the difference between a system call and a library call. Single user systems don't have to count system calls but a multi-user system needs to hoard system calls and treat them with respect since only one system call can be accomplished at a time (per cpu).

A sample is shown below of a profiled output for the ULTRIX ls command.

Partial Profile of the ULTRIX ls command

%time	cumsecs	seconds	calls	name	2.3	0.44	0.01	
22.7	0.10	0.10		mcount	0.0	0.44	0.00	2 _formatf
11.4	0.15	0.05	148	__doprnt	0.0	0.44	0.00	312 _strlen
9.1	0.19	0.04	361	_strcmp	0.0	0.44	0.00	92 _sprintf
6.8	0.22	0.03	269	_fcmp	0.0	0.44	0.00	92 _strcpy
6.8	0.25	0.03	92	_fmtentry	0.0	0.44	0.00	56 _printf
6.8	0.28	0.03	4	_qst	0.0	0.44	0.00	52 _malloc
6.8	0.31	0.03	1	_main	0.0	0.44	0.00	49 _readdir
4.5	0.33	0.02	360	__flsbuf	0.0	0.44	0.00	47 _cfree
4.5	0.35	0.02	47	_gstat	0.0	0.44	0.00	46 _cat
4.5	0.37	0.02	9	_morecore	0.0	0.44	0.00	3 _gtty
4.5	0.39	0.02	2	_calloc	0.0	0.44	0.00	3 _ioctl
2.3	0.40	0.01	50	_free	0.0	0.44	0.00	2 _bcopy
2.3	0.41	0.01	46	_savestr	0.0	0.44	0.00	2 _bzero
2.3	0.42	0.01	19	_sbrk	0.0	0.44	0.00	2 _isatty
2.3	0.43	0.01	10	_write	...			2 _qsort

[‡] UNIX is a trademark of A. T. & T.

[†] ULTRIX and MicroVAX are trademarks of Digital Equipment Corporation.

Consider what information could be obtained if the exact time-ordered sequence of calls was given. A complete trace of the algorithm being used would be found in this data. The traditional profile information gives a conglomeration of the usage often masking algorithmic data. The complete trace of a program however produces more data than one can imagine. To pare this down, consider only a trace of the system calls that a program uses. Although the complete picture is not available from a system call trace, it is often a strong indicator of what is really happening. This is the idea that led me to create a user level tool that gives a time-ordered list of system calls showing how the kernel is being used.

Minimizing system calls is one way to speed up the overall system. This minimizing of system calls reduces context switches, and rescheduling. It may also coalesce many calls into one call saving the overhead of multiple system calls. In asymmetric multi-processor environments, where one processor (the MASTER) does ALL system calls, minimizing system calls becomes a requirement since the slave processors may be idled for the duration of the call.

C programmers are often unaware of how the C library routines work causing routines to be misused. A time-ordered list of system calls gives inside information on the program is using the system facilities and allows the user to make judgments in the algorithm with respect to system resource usage. For example, there are many times (portability aside) that the library calls, *fread()* and *fwrite()* are faster than using the system calls, *read()* and *write()*.

2. System Call Tracer Design

The design of the system call tracer comes from three basic tenets. The first is to give easy access to trace data to normal users. This also means that multiple simultaneous users (currently configured for 16) are allowed. One simple command should be all that is needed to invoke the system call tracing facility. For example,

```
trace ls -l /etc /dev
```

is all that a user needs to type to run the command

```
ls -l /etc /dev
```

and get a trace.

The second goal is to minimize the impact upon the system. One extra test before and after the system call in the kernel is all that is allowed. The actual time to create the trace record and place it in a buffer has to be kept small. Processing time cannot be wasted formatting records nor can records be lost due to data flow problems.

The third goal is not to add any new system calls but to use the current system facilities. Tracing is basically creating data in the kernel and allowing a user to do a series of reads to get the data. UNIX already has that capability. The system call tracer is built upon 5 system calls and device entry points, namely, **open**, **read**, **ioctl**, **select**, and **close**. A device, */dev/trace*, was created in the *character device switch table* (*cdevsw*). The device is much like the device */dev/tty*. It provides a special entry into the kernel for the user to request, control, and obtain trace data from the kernel.

3. System Call Tracer Implementation

Following is a list of items needed to create the system call tracer:

1. A device, `/dev/trace`, and its entries in the `cdevsw`.
2. Two lines of C code within the kernel to test if tracing is enabled. The first line is before the system call is executed. If tracing is enabled, a call to a special routine to log the call and its arguments is made. The second line is after the system call returns. If tracing is enabled, a call to a special routine to log a completed call and its return values is made.
3. A new header file called `systrace.h` (shown below) containing the structures for the device `/dev/trace`.
4. A new source file called `sys_trace.c` containing the 5 device entry points, and the data formatting routines and buffer handling routines.
5. A user program, `trace`, that implements system call tracing for users. This program allows users to trace any command that can be typed from the shell.

The structure used in the kernel to keep track of system call tracing is:

```
#define TR_BUFSIZE 8192      /* must be <= MAXBSIZE */
#define FRACTION 3/5        /* % of full buffer when select wakes you up */
/* u.u_tracedev is the integer offset into tr_users[] */
#define u_tracedev u_XXX[0] /* spare field to use for now */
#define TR_USRS 16          /* number of simultaneous users */
#define TR_PIDS 16         /* number of pids to trace at a time */
#define TR_UIDS 16         /* number of uids to trace at a time */
#define TR_PGRP 16         /* number of pgrps to trace at a time */
#define TR_SYSC 16         /* number of syscalls to trace at a time */
int traceopens;            /* count the opens to enforce limit */
int tracelost;             /* count of lost records from buffer overflow */
int tracebsize = TR_BUFSIZE; /* trace buffer size */
struct trace {
    int pids[TR_PIDS];      /* pids we are tracing for this user */
    int uids[TR_UIDS];      /* uids we are tracing for this user */
    int sysc[TR_SYSC];      /* syscalls we are tracing for this user */
    int pgrp[TR_PGRP];      /* pgrps we are tracing for this user */
    int traceflag;          /* what to trace -- see flags below */
    int open;               /* is this pseudo-device open */
    struct buf *bp;         /* buffer pointer */
    int uid, ruid;          /* privilege controls set at open */
    struct proc *tr_proc;   /* proc addr for select */
} tr_user[TR_USRS];        /* one structure per user */
```

The global variable `traceopens` counts the number of opens done on the trace device. It is the variable tested to see if tracing has been invoked. The global variable `tracelost` counts lost records. The global variable `tracebsize` contains the size of the trace record buffers. It can be changed by `adb-ing` the kernel.

Originally, the code was written so that a user could trace system calls via several different attributes. They are: *user id*, *process id*, *process group*, and *system call number*. The most useful of these has been the process id and the process group. It turns out there is a security hole if any user (including root) is allowed to trace any other user's processes. To demonstrate the problem, suppose a user (the

tracer) is allowed to trace someone's (the victim) login shell process group. The tracer just waits for the victim to run `rlogin somevax -l root`, and magically the root password for the machine somevax will appear in the trace record soon after the victim types it in. Even worse, if someone were allowed to trace all read system calls, they could get all the passwords for anyone logging in! A secure version of the system call tracer exists and it only allows a user whose real uid and effective uid are the same to trace his or her own processes. It forbids the tracing of any process that is a setuid or setgid process so that even root could not trace the rlogin process. In a development environment, the ability to trace anything is actually used to trace daemons and login processes for engineering purposes. This is accomplished by just taking out a few lines of security testing code.

The system call tracing facility is invoked each time the device `/dev/trace` is opened. The user who did the open gets assigned one of the `struct trace` structures to control the trace. The structure contains the attributes of what is being traced by process id, user id, process group, or system call number. It keeps a pointer to the buffer of trace records, and keeps track of security information and process information related to the trace.

A critical number in the implementation is the **FRACTION** of a full buffer at which the `select()` system call wakes up the process that is reading the trace records. It is currently set at 60% and works well while consuming very little cpu time. In fact, a trace was done that wrote and read an 1 megabyte file ten times. This trace was run on a MicroVAX II and produced a 1.7 megabyte trace record in which NO trace records were lost. On busier machines your mileage may vary.

4. System Call Traces and Fixes

Now for some of the goodies. A few detailed cases will be presented so that that essence of system call tracing may be seen. A big (but not the entire) list of things that we found wrong and that should be fixed will be given. Note that all of the tracing was done on a pre-Version 1.2 of ULTRIX. Some of these fixes are already in Version 1.2 of ULTRIX and others are targeted for future releases. Most are applicable to other versions of UNIX.

4.1. trace ps -ax

TIME STAMP PID SYSCALL ARGS.....

```
... who do we execute today
961.430000 1789 C execve ( "ps", 22724/ 0x58c4, 23892/ 0x5d54 )
961.440000 1789 R execve Error 2
961.440000 1789 C execve ( "./ps", 22724/ 0x58c4, 23892/ 0x5d54 )
961.450000 1789 R execve Error 2
961.450000 1789 C execve ( "/usr/rr/bin/ps", 22724/ 0x58c4, 23892/ 0x5d54 )
961.460000 1789 R execve Error 2
961.480000 1789 C execve ( "/usr/local/bin/ps", 22724/ 0x58c4, 23892/ 0x5d54 )
961.490000 1789 R execve Error 2
961.590000 1789 C execve ( "/usr/ucb/ps", 22724/ 0x58c4, 23892/ 0x5d54 )
961.600000 1789 R execve Error 2
961.600000 1789 C execve ( "/bin/ps", 22724/ 0x58c4, 23892/ 0x5d54 )
... execve's can be costly if your path is not right
... or you use the Bourne shell
... open vmunix to get namelist
961.880000 1789 C open ( "/vmunix", 0, 438/ 0x1b6 )
961.890000 1789 R open 8
... read in the namelist using nlist.c
```

```

964.540000 1789 C read ( 8, 0x11004, 4096/ 0x1000 )
964.570000 1789 R read 4096 byte(s) : "_rpb?_scb?_Xpassrel?_Xmachcheck"
964.580000 1789 C lseek ( 8, 0, 1 )
964.580000 1789 R lseek 0x5e962
964.690000 1789 C lseek ( 8, 0, 1 )
964.690000 1789 R lseek 0x5e962
964.690000 1789 C lseek ( 8, 0, 1 )
964.700000 1789 R lseek 0x5e962
964.700000 1789 C lseek ( 8, 0, 1 )
964.700000 1789 R lseek 0x5e962
... 70 more lseeks later
965.480000 1789 C read ( 8, 0x11004, 4096/ 0x1000 )
...

```

Stdio does *fread()* and *fseek()* in its buffer but must do an *lseek()* each time to keep its offset in sync. The fix is to add a file pointer to the stdio file structure thus eliminating the *lseek()* when *fseek()*ing in an internal buffer. Rewriting the nlist algorithm, and doing the trace gives the following comparison:

Total Number of System Calls

	"ps -ax"
Old ps	662
New ps	395

We eliminate 267 system calls (most of them lseeks).

4.2. trace ls -l /dev /etc

```

TIME STAMP  PID  SYSCALL          ARGS.....
... after a bunch of lstats for the -l option
... ls is now formatting the output
134.330000 1685 C gettimeofday ( 0x7ffdfef8, 0x7ffdfef0 )
134.340000 1685 R gettimeofday 0
134.350000 1685 C gettimeofday ( 0x7ffdfef8, 0x7ffdfef0 )
134.350000 1685 R gettimeofday 0
... 20 more gettimeofday calls later
... ls writes its output for the directory / and
... does the rest of the args

```

Each file's time has to be converted to the familiar character representation "Jul 22 19:53" and this is done with *ctime(time)*. However, the only reason *ctime()* calls *gettimeofday()* is for the timezone! Thus, the fix for *ctime()* is to call *gettimeofday()* once and remember the timezone. For this example, 119 calls to *gettimeofday()* are reduced to 1 call.

4.3. trace fprintf

```

#include <stdio.h>
main()
{
    fprintf(stderr,"This is a string");
}

```


TIME STAMP PID SYSCALL ARGS.....

... running the above simple program gives

```

987.570000 1675 C write ( 2, "T", 1 )
987.570000 1675 R write 1
987.580000 1675 C write ( 2, "h", 1 )
987.590000 1675 R write 1
987.600000 1675 C write ( 2, "i", 1 )
987.600000 1675 R write 1
987.610000 1675 C write ( 2, "s", 1 )
987.610000 1675 R write 1
987.620000 1675 C write ( 2, " ", 1 )
987.630000 1675 R write 1
987.640000 1675 C write ( 2, "i", 1 )
987.640000 1675 R write 1
987.650000 1675 C write ( 2, "s", 1 )
987.650000 1675 R write 1
987.660000 1675 C write ( 2, " ", 1 )
987.670000 1675 R write 1
987.680000 1675 C write ( 2, "a", 1 )
987.680000 1675 R write 1

```

Unbuffered I/O is written ONE character at a time. This is a deficiency in stdio. The affected routines are *fwrite()*, *fputs()*, and *fprintf()*.

4.4. trace rcp bigfile a:/tmp

TIME STAMP PID SYSCALL ARGS.....

... call *gethostbyname()* for local and remote machines

```

660.040000 1763 C open ( "/etc/hosts", 0, 438/ 0x1b6 )
660.050000 1763 R open 5
660.050000 1763 C fstat ( 5, 0x7ffddb70 )
660.050000 1763 R fstat 0
660.060000 1763 C read ( 5, 26628/ 0x6804, 4096/ 0x1000 )
660.120000 1763 R read 1774 byte(s) : "127.0.0.1 localhost 98.0.0.1 li"
660.140000 1763 C close ( 5 )
660.190000 1763 R close 0
660.200000 1763 C open ( "/etc/hosts", 0, 438/ 0x1b6 )
660.200000 1763 R open 5
660.210000 1763 C fstat ( 5, 0x7fffd8e8 )
660.210000 1763 R fstat 0
660.210000 1763 C read ( 5, 26628/ 0x6804, 4096/ 0x1000 )
660.250000 1763 R read 1774 byte(s) : "127.0.0.1 localhost 98.0.0.1 li"
660.280000 1763 C close ( 5 )
660.280000 1763 R close 0
... read/write data
662.510000 1763 C read ( 6, 0x7fffd814, 1024/ 0x400 )
662.570000 1763 R read 1024 byte(s) : "1 This is a silly file to read "
662.580000 1763 C write ( 5, "This is a silly file to read so", 1024/ 0x400 )
662.590000 1763 R write 1024/ 0x400
662.600000 1763 C read ( 6, 0x7fffd814, 1024/ 0x400 )
662.600000 1763 R read 1024 byte(s) : "2 This is a silly file to read "

```

Changing the read and write sizes to 4096 characters, and doing the trace gives the following comparison:

Total Number of System Calls

	"rcp bigfile a:/tmp"
Old rcp	271
New rcp	143

We eliminate 128 read and write system calls.

4.5. cc -c file.c

TIME STAMP PID SYSCALL ARGS.....

... my cc for pipes

996.520000 195 C execve ("/usr/staff/rr/bin/cc", 22744/ 0x58d8, 23916/ 0x5d6c)

... set up the pipes

996.920000 198 C execve ("/lib/cpp", 11396/ 0x2c84, 0)

997.640000 198 C read (5, 23768/ 0x5cd8, 4096/ 0x1000)

996.940000 197 C execve ("/lib/ccom", 11412/ 0x2c94, 0)

998.140000 197 C read (0, 0x36c04, 2048/ 0x800)

998.840000 197 C read (0, 0x36c04, 2048/ 0x800)

996.890000 196 C execve ("/bin/as", 11420/ 0x2c9c, 0)

998.280000 196 C read (0, 0x10c38, 8192/ 0x2000)

999.220000 196 C read (5, 0x19c04, 4096/ 0x1000)

...

As some of you might know the C compiler can be used as a completely pipelined process. Normally each phase (*cpp*, *ccom*, *c2*, and *as*) is run as a separate process. Data is stored in intermediate files as communication from process to process. However, since these tools were written as true UNIX tools they read and write their standard input and output. Hence, an equivalent way to compile instead of *cc -c file.c* is:

`/lib/cpp file.c | /lib/ccom | as -o file.o`

By accident, I actually traced a *cc* command and got a trace similar to the one above. I noticed that */lib/ccom* was reading the pipe at only 2048 characters and I/O on pipes in ULTRIX is supposed to be 4096 characters!

4.6. A Long List of Utilities in Need of Work

- *Sed* reads and writes 1024 characters.
- *Adb* does single character reads and writes.
- *Sh* does single character reads of *.profile* at login time.
- *Libterm* routines read */etc/termcap* 1024 characters at a time.
- *Ed* reads 1 character at a time from *stdin*.
- *Head* writes one line at a time.
- *Ld* does a LOT of unnecessary seeking.
- *More* reads files 1024 characters at a time.
- *Tar* reads from and writes to the disk in 512 byte buffers!

- *Sh* reads scripts 64 characters at a time.
- In a shell script like *spell*, using full path names for the commands would prevent path name searches by the Bourne shell and save 5-10% of the *execve* system calls it does just to find the right program to execute. Static scripts should use full path names or variables to invoke programs.
- Out of date system calls exist in many programs. Take the example:

```
if (!access("file"))
    close(creat("file"));
open("file", mode);
```

Of course this code is dated since *open()* now has three arguments. The code can be replaced with:

```
open("file", O_CREAT, mode);
```

In this case 4 system calls become one. Many other examples of out dated code exist throughout the system tools.

- Tracing a shell like *sh* or *csh*, you see it does closes on file descriptors from 3 to 63. In fact there are many programs that contain code like:

```
for ( i=3; i < NOFILE; i++ ) close(i);
```

On ULTRIX, **NOFILE** is now 64 (and growing!?). This is getting to be a serious load since closes are done on every fork and exec! The last 44 or more system calls to close return the error **Illegal File Descriptor**. Since file descriptors are inherited through the fork/exec process, once a program does an exec it loses the data on the highest file descriptor in use and so only the system knows what files are open. A system call that returns information about process data would solve this problem.

5. Future Work

Many other things happen to processes in UNIX. Tracing data about signal events, context switches, sleep and wakeup, scheduling events and I/O events are prominent on the list for data to obtain. Certainly we are very interested in looking at more detailed file I/O data similar to that obtained in the recent Berkeley studies. Data on the buffer cache and device interactions would also be of interest.

6. Acknowledgements

The author wishes to thank Carolyn Zappala without whom much of the tracing would not have been done. I did the easy 5% of the tracing and she did the hard 95%. The author also wishes to give credit to S. Zhou for the Berkeley File tracing code and the valuable conversations about the file tracing work at UC Berkeley. Joseph D. Simonetti was kind enough to place his system call tracer for 4.2BSD on the USENET. I borrowed his ideas for the formatting routine and streamlined it for speed. I also thank all the reviewers for their helpful comments and the ULTRIX Documentation Group for the help in making this paper more presentable.

Two papers at one conference is too much. I am grateful for the support of my wife and all my friends in this endeavor.

7. References

The references that follow serve as good entry points into the literature. The interested reader would find many hours of enjoyable reading by tracking down these papers and those that they in turn reference.

- Leffler, Samuel J., "Measuring and Improving the Performance of 4.2BSD", Technical Report UCB-CSD-84-218, 1984.
- McKusick, Marshall Kirk, "Performance Improvements and Functional Enhancements in 4.3BSD", Technical Report UCB-CSD-85-245, 1985.
- Ousterhout, John K., Da Costa, Herve, Harrison, David, Kunze, John A., Kupfer, Mike, Thompson, James G., "A Trace-Driven Analysis of the UNIX 4.2BSD File System", Technical Report UCB-CSD-85-230, 1985.
- Simonetti, Joseph D., "A System Call Trace Facility", Technical Report SUNY 001, 1985.
- Zhou, Songnian, Da Costa, Herve, Smith, Alan Jay, "A File System Tracing Package for Berkeley UNIX", Personal Communication, 1985.

A New Virtual-Memory Implementation for Unix

Edward W Sznyter¹

Patrick Clancy

James Crossland²

Graphics Workstation Division

Tektronix, Inc.

Wilsonville, Oregon

ABSTRACT

The UTek³ operating system, Tektronix' enhanced version of Berkeley 4.2 Unix⁴ running on Tektronix 6000-series workstations, includes a redesigned and enhanced virtual memory system which allows a process great flexibility in controlling its address space. New virtual memory features include shared memory among processes with differing access privileges, memory allocation that is non-contiguous within the virtual space, copy-on-write sharing of memory, and direct mapping of physical memory. These features can be used to support various types of resource management, such as shared libraries. Some existing applications which use these features are described, along with potential for future use.

1. INTRODUCTION

UTek³ is Tektronix' name for an enhanced version of Berkeley 4.2 Unix⁴ which runs on Tektronix 6000-series workstations. In it's latest released form (2.2), the UTek kernel is built around a new virtual memory implementation which differs substantially from 4.2. This new implementation supports sharing of memory among processes with differing access privileges, non-contiguous segments within the address space, copy-on-write sharing of data pages, creation of zero-fill-on-demand pages, and direct mapping to physical memory. The design is loosely based on the set of VM system calls which were documented [6] but not implemented for Berkeley release 4.1c. The design was also influenced conceptually by MULTICS [1,3] and TENEX/Tops-20 [2,4].

The new virtual memory system is designed to give the user control over the address space on a page-by-page basis. However, it is completely transparent to programs that don't use the new system calls, with the exception of those that rely on unique side-

¹Current address: Mathematics Dept., University of Washington, Seattle, Washington.

²Current address: Computer Science Dept., University of Utah, Salt Lake City, Utah.

³UTek is a trademark of Tektronix.

⁴Unix is a trademark of AT&T Bell Laboratories.

effects of *vfork*.

2. DESIGN CONCEPTS

The fundamental virtual memory management structure within the kernel is the *segment*. A segment may be thought of as a group of pages which are always contiguous in virtual space. For example, the text area of a newly exec'ed process may be considered to be a segment. A segment has its own page table stored on disk, which maintains information about pages which are not otherwise referenced from in-core page tables.

Each process has a list of *mappings* from which its address space is built. A mapping is like a window from a process' virtual space onto some part of a segment's virtual space. It need not map the segment's first page. Much effort was put into making this translation process as efficient as possible, since the kernel uses the new routines for its normal manipulation of process space.

A standard process (read-only text, demand paged) initially starts with mappings corresponding (by convention) to the four compiler-generated areas of text, initialized data, uninitialized data, and stack.

3. FUNCTIONAL INTERFACE

Three new system calls are provided for memory sharing and address space rearrangement (see Appendix):

mmap(pid, fromaddr, toaddr, len, prot, share)

mmap is used to set up a new mapping for the current process, with specified protection and sharing attributes. The area from *toaddr* to *toaddr+len* must be free, or an error (EMCOLLIDE) is returned. The mapping at *fromaddr* must have protections which allow the new mapping. Protections are based on access by owner, process group, user group, and world.

If *pid* is a valid process id, the area at *toaddr* will contain a copy of the memory at location *fromaddr* of the designated process. If the argument *share* is PRIVATE, the pages are marked copy-on-write, and a new copy will be made if any attempt is made to modify a page. If *share* is SHARED, the pages are shared and modifications by one process are visible to the other.

If *pid* is SELF, the mapping is made from the current process. If *pid* is FZERO, *fromaddr* and *share* are ignored, and the new area is set up so that any reference will generate a page fault, and a zero-filled page will be created at that location.

If *pid* is PHYSICAL, references to the area will access the specified physical locations. This type of mapping requires root user privileges.

mremap(fromaddr, toaddr, len, prot)

mremap is used to move a mapped area within the address space, possibly with new protections. This may include multiple mappings, parts of mappings, and unmapped areas. The new area will have the same data with the given protection. To change the protection of an area, memory is moved onto itself. The entire new area must be free after the old area is gone, or no action is taken and an error code (EMCOLLIDE) is returned.

munmap(addr, len)

munmap is used to remove the mappings in the specified area of the address space. A subsequent reference to the area will generate a fault (EFAULT). The area may span several different mappings and holes. Note that anything, including text, may be unmapped.

The copy-on-write mechanism is also used in the implementation of *fork*, superceding the *vfork* of 4.2. (In UTek 2.2, *vfork* and *fork* execute the same code in the kernel). The data area of the child process is created as a private mapping of the corresponding area (all writable mappings) of the parent. The text area is created as a shared mapping to the text area of the parent.

4. IMPLEMENTATION

4.1 Segments and Mappings

Two new data structures represent mappings and segments. Each segment has an associated page table kept on disk. This segment page table is not directly used by any executing process. A mapping is said to be *applied* when it is set up with in-core pages and page table entries, so that the process can execute and use the mapping. When a mapping is not applied, its pages (if not also part of another applied mapping) are in swap space and are located by obtaining their disk addresses from the segment's page table. Individual pages of an applied mapping may still be paged out to their swap space location by the pageout daemon (proc[2]). A process is swapped out by swapping out all of its mappings, swapping out its *user* structure, and freeing its page tables.

Swap space for a page is allocated when the page is allocated (eg.: by the first fault on the page). A bit map is used to keep track of free page blocks within the swap space. The allocation algorithm maintains a current pointer into the bit map, which it uses as the starting place on each invocation to search for the next free block.

Each mapping refers to a segment, with an associated offset and length within the segment's space, and an offset within the address space of the process that owns the mapping. Multiple mappings to the same area of the same segment correspond to sharing of real memory, and use the same real pages. Text is initially mapped this way, and *mmap* may create other sharing of segments.

4.2 Page Tables

Page table entries contain new information to support the VM system. This includes a three-bit type field which indicates the current location/type of the page: in-core, in swap space, in a file, privating (see below), zero-fill, or physical. Only in-core or privating pages may be valid. For a page on the disk which belongs to an applied mapping (the only kind that have page tables), the page frame field contains the current disk address instead of the physical page frame number. In addition, page table entries contain a bit which indicate whether the page is privated; ie, whether another page is privating (copy-on-write from) this page.

Allocation and management of page tables for in-core processes is virtually unchanged from the 4.2 version; page tables expand up and down to cover the entire area between the lowest and highest mapped text and data addresses (and similarly for the stack). This implies that some memory may be wasted due to its being used for page tables covering areas between valid process mappings; in practice, this loss is not significant. Paging of user page tables (not currently implemented in UTek) is one approach to addressing this issue.

4.3 Copy On Write

When an area of a segment is private, both a new mapping and new "private" segment are created for the privating process. In-core pages of both the privating and the private segments are made valid read-only. A "private" segment may itself be private, possibly resulting of a chain of "privating-from-private" mappings and segments, with only the first segment in the chain representing "real" pages. Although all the mappings in the chain share the pages for reading, the mappings and ptes for this first segment control the status of the pages (eg.: valid in-core or swapped out) at any given time.

A segment is freed when there are no shared or private mappings which refer to it. A segment therefore has a lifetime which is completely independent from the lifetime of the process which initially caused it's creation.

4.4 Page Replacement

The use of the global clock replacement algorithm implemented in 4.2 is retained in UTek 2.2.

4.5 Pre-fetch

On a page fault, a group (*kluster*) of up to eight pages (on a 6130 workstation) may be brought in to memory. The group size is determined by the number of pages in the appropriate mapping, starting with the faulting page, which reside at physically contiguous locations in swap or file space on the disk.

4.6 Resident Set Size

Proper tracking of the number of resident (in-core) pages for a given process involves keeping a separate resident set size counter with each segment and mapping, as well as the process. When a page is added to or deleted from a segment, the resident set sizes of the segment, all mappings using the page, and all their owning processes, are modified. Segment resident set size is the only accurate measure of actual resident pages. However, the swapper cannot directly use segment resident set sizes in deciding eligibility for swapping, because the pages covered by the process mappings may not correspond to the actual resident pages of the segment. Therefore, the swapper uses process resident set size, which is the sum of the sizes for all the mappings of the process, but may not reflect true gain or loss on a swap.

4.7 Brk

The UTek kernel necessarily retains the idea of a "break" at the end of the data segment, for compatibility. To integrate this mechanism with the new VM model, the break location is considered to be associated with the particular mapping which is first created for the segment. If the mapping is moved or resized via the use of one of the new system calls, the break is reset to stay at the end of the mapping.

5. CURRENT APPLICATIONS

Several applications and utilities have been designed to use the new VM features of UTek 2.2. A prototype implementation of shared libraries is also in progress.

5.1 Shared Libraries

Every Unix system provides several standard libraries consisting of functions which can be linked to user programs prior to run-time. It would be advantageous to have all running programs access a single in-core copy of this library code, rather than using up memory and disk resources to give each program it's own private copy.

Shared libraries can be implemented relatively easily using the new mapping facilities. The prototype design has the following desirable characteristics: (1) there is minimal added run-time overhead to share libraries; (2) installation of a new version of a library does not require re-compilation or re-linking of programs which use the library; (3) the premature termination of a process providing library services will not cause the termination of other processes using the library. (4) no modifications need be made to the existing compiler, loader (*ld*), other tools, or the libraries themselves; (5) no further kernel involvement (beyond the new mapping facilities) is required.

Shared library code and data are linked into a program which runs as a library server daemon process. This program includes a jump table which consists of a series of branch instructions to library functions. Library code is loaded after the jump table, and library data (both initialized and bss) follows the code page-aligned at a fixed offset from the beginning (to allow more code in later revisions without changing data load offsets). The jump table and libraries are relocated at link time to start at a high address, which will be above the base of the user stack.

When the server starts up, it first uses *mmap* to change the access permissions on it's entire mapped address space to readable-by-world. Then, an information block is written to */etc/slib*, consisting of the server's process id, the starting virtual address of the data segment, and the size of the text and data segments.

User programs intended to use the shared library must be linked at compile-time with special run-time start up code. Also, they must be relocated during the final load (by *ld*) to reference appropriate jump table locations in all subroutine calls to library functions, and to reference any global library data at its correct loaded address. The standard available "incremental loading" capability of *ld* is utilized. This loader option allows an object file (in this case, one representing the jump table and global library data) to be specified as supplying symbol table information only; the user program is relocated using the correct jump table offset addresses. (Note that none of the jump table or library code or data is actually loaded with the user program).

At run time, the start-up code which was linked with the user program has to map the jump table and library code and data from the server into the user's address space. It first reads the information block from */etc/slib*. Then the user stack pointer is decremented so that the new stack base will leave enough room for the jump table and libraries above the stack (the amount of space needed is calculable from the information block plus the known fixed size of the jump table). The initial stack pages are remapped down to the lower addresses. The jump table and library text from the server are now mapped read-only and shared to the area above the stack. The data area from the server is mapped writable and *private* above the library text. Because a private map is made of library data, the user process will get it's own private copy of library data pages on the first write.

5.2 Basic

The Tektronix implementation of ANSI Basic uses shared memory to implement asynchronous I/O, and uses the general mapping facilities to organize allocation of different data types. For I/O, a child process which shares data and control buffers with the parent Basic process is created. The child waits for I/O completion while the parent remains unblocked.

For memory allocation, the implementation foregoes entirely any use of *brk/sbrk*; instead, *mmap* is used to set up separate disjoint data areas for strings, arrays, scalars, and several internal data types that are scattered through the address space. Each of these areas has a bit vector at the end which is used to keep track of which pieces of the area are free or in-use. When an area has no free space left, it is expanded and the bit vector is remapped to remain at the end of the new larger area. This approach would be impractical under the *brk/sbrk* allocation model.

5.3 Kernel Utilities

Certain utilities such as *ps*, which examine large kernel data structures, can be profitably re-implemented to use the *mmap* facilities. The standard implementation of *ps* uses the kernel virtual memory special file (*/dev/kmem*) to read the contents of the kernel *proc* table into an internal buffer, and then examine table entries to determine the status of processes. This approach involves the overhead of an I/O operation (although it does not go to disk), and the copying of the entire table into *ps*'s buffer.

The *ps* which is supplied with UTek 2.2 has been re-implemented to use *mmap*. The */dev/kmem* file is accessed only to get the kernel virtual address of the *proc* table, and then to fetch the appropriate kernel page table entry to determine the table's starting physical address. Then *mmap* is used to map part of *ps*'s space to this physical memory, avoiding any copying.

5.4 Datek

Datek is an Ingres-like database system which is used internally within Tektronix. The system consists of a server process and separate client processes for each of the users. A client process serves as a user interface and does the necessary formatting of input and output. The server process is responsible for receiving client requests for data, and for updating data records which the client has locked. For each new client process the server maps a new set of control and data buffers from the client's address space into its own address space. Communication between the clients and the server is performed through these buffers. This centralization of database file access allows record locking and access serialization to be confined to a single process, which simplifies correct design and implementation of the system.

6. POTENTIAL USES

One potential use for the new virtual memory features is to provide a base for algorithms which improve the performance of AI language implementations. Functional or object based interpreted languages typically need to structure their address spaces differently than Algol-based compiled languages which conform to the text/global-data/heap-data/stack model. The ability to create arbitrary mappings in the space allows such structuring to be done without constraints being imposed by the operating system.

The fact that physical memory can be mapped into privileged user tasks creates the possibility of efficient user mode device drivers, and other types of user mode servers. Language systems such as Smalltalk-80 [5], which require direct access to hardware display memory, can use *mmap* to gain access anywhere in their virtual space. Also, hardware configuration information can be made available to such systems at known physical locations. Drivers and servers (eg., to manage a display and mouse) which are part of such language processing systems may therefore be implemented without extensive operating system additions or reconfiguration.

In general, a memory management scheme which copies page-sized structures and operates in physical memory can be redesigned to simply remap these structures and

avoid the copying if running in virtual memory. One area of future investigation will involve the discovery of ways in which the new VM features of UTek can be used to redesign suitable memory management and garbage collection algorithms which profit from the remapping capability.

A general problem encountered in language environments concerns the need to manage multiple large blocks of unrelated data (as in the Basic example cited above). In a standard virtual memory environment such programs must compromise between allocating very large segments of memory for their various data types and minimizing the resources needed to run the program. Programmers try to set the data sizes so that they are adequate for the average program, but when larger problems are attempted, the data areas have to be expanded. This requires moving data or linking multiple disjoint segments. The former incurs high overhead at the time of expansion and the latter means more overhead each time the data structures are referenced. These programs experience considerable performance degradation when a larger than the expected size of problem is attempted.

With the new mapping facilities the programmer is free to spread data structures over the entire address space, reducing the possibility of a collision between two unrelated data areas. In fact, on smaller systems it is possible that the program will run out of some critical resource, such as swap space, before a collision between two disjoint data areas occurs. If a collision happens, *mremap* can be used to move an entire segment. This requires that only the page table entries be copied and not the data they point to. The performance of such programs are thus not degraded due to internal memory management when larger problems are attempted.

7. PERFORMANCE

Overall system performance of standard 4.2 virtual memory is 5-10% better than the new VM implementation for UTek 2.2. This is a result of many factors, including: (1) increased amounts of system I/O to read and write segment page tables on disk; (2) increased context switch activity as a result of the increased I/O load; (3) frequent allocation and deallocation of paging blocks in swap space; (4) the increased number of kernel data structures to be referenced on creation and destruction of processes, and whenever page table entries are modified. In general, the slight performance loss has not been noticeable to users, and we feel it to be more than offset by the increased flexibility of the system and opportunities for performance gains at the applications level through use of the mapping facilities.

8. CONCLUSION

The UTek enhanced virtual memory design provides great flexibility to users in determining how process address spaces are configured and shared. For applications consisting of multiple cooperating processes, shared memory is often the desired paradigm for inter-process communications. For very large memory-intensive applications, the copy-on-write capability makes process activation efficient by eliminating unnecessary copying of text and data segments from parent to child processes. System wide sharing of resources such as libraries is feasible without modifications to existing tools and without bringing the resource under direct control of the kernel. Language processors can use the mapping capabilities to more easily and efficiently manage the run-time environment without special kernel services. We feel that we have only made a small beginning in discovering ways to effectively utilized the shared-memory model and address space control extensions to Unix.

Acknowledgements

Mike Zuhl contributed direction as manager of the project. Andrew Klossner incorporated the new features into Tektronix ANSI Basic.

References

- [1] Bensoussan, A., Clingen, C. T., and Daley, R. C., "The Multics Virtual Memory Concepts and Design", *Communications of the ACM* (May 1972).
- [2] Bobrow, Daniel G., Burchfiel, Jerry D., Murphy, Daniel L., and Tomlinson, Raymond S., "TENEX, a Paged Time Sharing System for the PDP-10", *Communications of the ACM* (March 1972).
- [3] Daley, Robert C. and Dennis, Jack B., "Virtual Memory, Processes and Sharing in MULTICS", *Communications of the ACM* (May 1968).
- [4] *DECSYSTEM-20 Monitor Calls Reference Manual*, Digital Equipment Corporation, September 1978.
- [5] Goldberg, A. and Robson, D., *Smalltalk-80*, Addison-Wesley, 1983.
- [6] *Unix Programmer's Manual, Seventh Edition*, Dept. of Electrical Engineering and Computer Science, Univ. of Calif, Berkeley, March 1983.

NAME

mmap – map pages of memory

SYNOPSIS

```
#include <sys/types.h>
#include <sys/mman.h>

mmap(pid, fromaddr, toaddr, len, prot, share)
int pid;
caddr_t fromaddr, toaddr;
u_int len, prot, share;
```

DESCRIPTION

The mapping routine **mmap** allows a process to access areas of other processes through its own address space. It causes the calling process' pages starting at *toaddr* and continuing for *len* bytes to map onto the process with id *pid*, starting at the object's pages *fromaddr*.

If *pid* is *M_SELF*, an area of the process is mapped to itself. If *pid* is *M_PHYS*, an area of the process is mapped to physical memory (in which case *share* is ignored). If *pid* is *M_ZFILL*, an area of the process is made zero filled (in which case *fromaddr* and *share* are ignored).

If the parameter *share* is true, both mappings will share the same memory. Otherwise, a *private* copy of the area is made, and changes through one mapping are not visible through the other.

PRIVATE	make a private copy for the new mapping
SHARED	share the area between the mappings

The parameter *prot* specifies the accessibility of the pages through the new mapping. Read and write access may be given on the basis of processes of the same user, same process group, same group, and world. A process may also protect its pages against itself. The protection for a page is specified by *or'ing* together the following values.

M_R_SELF	read, process
M_W_SELF	write, process
M_R_USER	read, user
M_W_USER	write, user
M_R_PGROUP	read, process group
M_W_PGROUP	write, process group
M_R_GROUP	read, group
M_W_GROUP	write, group
M_R_WORLD	read, world
M_W_WORLD	write, world

Note that the protection is associated with the mapping, and not with the actual memory.

If the process must change the protection of a mapping, it may map the area to itself, with the new protection. Doing this with *share* cleared will disassociate the area with all other mappings.

The *toaddr*, *fromaddr* and *len* parameters must be multiples of the system cluster size (found using the *getpagesize(2)* call).

DIAGNOSTICS

Mmap will fail when one of the following occurs:

[EINVAL]

An address is not on a cluster boundary.

[EMCOLLIDE]

Portions of the new area are already mapped.

[EMRANGE]

An area is outside the possible user's address space or includes part of the uarea.

[EACCES]

The required permissions (for reading and/or writing) are denied for the named file or area of a process.

[ESRCH]

No process can be found corresponding to the specified *pid*.

[EPERM]

The area of the object to be mapped is protected against the desired operation.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

getpagesize(2), *mremap(2)*, *munmap(2)*.

NAME

mremap – remap pages of memory

SYNOPSIS

```
#include <sys/types.h>
#include <sys/mman.h>

mremap(fromaddr, toaddr, len, prot)
caddr_t fromaddr, toaddr;
u_int len, prot;
```

DESCRIPTION

Mremap causes the process pages starting at *fromaddr* and continuing for *len* bytes to be mapped to the address *toaddr*. The parameter *prot* specifies the accessibility of the newly mapped pages (see *mmap*(2)).

The *fromaddr*, *toaddr*, and *len* parameters must be multiples of the system page size (obtained with the *getpagesize*(2) call), which may be larger than the underlying hardware page size.

DIAGNOSTICS

[EINVAL]

An address is not on a cluster boundary.

[EPERM]

The area of the object to be mapped is protected against the desired operation.

[EMCOLLIDE]

Portions of the new area are already mapped. This check is made as if the old area were gone, so overlapping moves work.

[EMRANGE]

An area is outside the possible user's address space or includes part of the uarea.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

fmap(2), *getpagesize*(2), *mmap*(2), *mremap*(2), *munmap*(2).

NAME

`munmap` – unmap pages of memory

SYNOPSIS

```
#include <sys/types.h>
#include <sys/mman.h>

munmap(addr, len)
caddr_t addr;
u_int len;
```

DESCRIPTION

Munmap causes the process pages starting at *addr* and continuing for *len* bytes to be removed from the legal address space of the process.

DIAGNOSTICS

[EINVAL]

addr is not on a cluster boundary or *len* is not a multiple of the pagesize.

[EMRANGE]

The area to be unmapped is outside the possible user's address space or includes part of the uarea.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

fmap(2), *getpagesize(2)*, *mmap(2)*, *mremap(2)*.

Mach: A New Kernel Foundation For UNIX Development¹

Mike Accetta, Robert Baron, William Bolosky, David Golub,
Richard Rashid, Avadis Tevanian and Michael Young
Computer Science Department
Carnegie Mellon University
Pittsburgh, Pa. 15213

Abstract

Mach is a multiprocessor operating system kernel and environment under development at Carnegie Mellon University. Mach provides a new foundation for UNIX development that spans networks of uniprocessors and multiprocessors. This paper describes Mach and the motivations that led to its design. Also described are some of the details of its implementation and current status.

1. Introduction

Mach² is a multiprocessor operating system kernel currently under development at Carnegie-Mellon University. In addition to binary compatibility with Berkeley's current UNIX³ 4.3 bsd release, Mach provides a number of new facilities not available in 4.3:

- Support for multiprocessors including:

- provision for both tightly-coupled and loosely-coupled general purpose multiprocessors and
- separation of the process abstraction into *tasks* and *threads*, with the ability to execute multiple threads within a task simultaneously.

- A new virtual memory design which provides:

- large, sparse virtual address spaces,
- copy-on-write virtual copy operations,
- copy-on-write and read-write memory sharing between tasks,
- memory mapped files and
- user-provided backing store objects and pagers.

¹This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4864, monitored by the Space and Naval Warfare Systems Command under contract N00039-85-C-1034.

²Mach is *not* a trademark of AT&T Bell Laboratories (so far as we know).

³UNIX, however, *is* a trademark of AT&T Bell Laboratories.

- A capability-based interprocess communication facility:
 - transparently extendible across network boundaries with preservation of capability protection and
 - integrated with the virtual memory system and capable of transferring large amounts of data up to the size of an address space via copy-on-write techniques.
- A number of basic system support facilities, including:
 - an internal adb-like kernel debugger,
 - support for transparent remote file access between autonomous systems,
 - language support for remote-procedure call style interfaces between tasks written in C, Pascal and CommonLisp.

The basic Mach abstractions are intended not simply as extensions to the normal UNIX facilities but as a new foundation upon which UNIX facilities can be built and future development of UNIX-like systems for new architectures can continue. The computing environment for which Mach is targeted spans a wide class of systems, providing basic support for large, general purpose multiprocessors, smaller multiprocessor networks and individual workstations (see figure 1-1). As of April 1986, all Mach facilities, with the exception of threads, are operational and in production use on uniprocessors and multiprocessors by both individuals and research projects at CMU. In this paper we describe the Mach design, some details of its implementation and its current status.

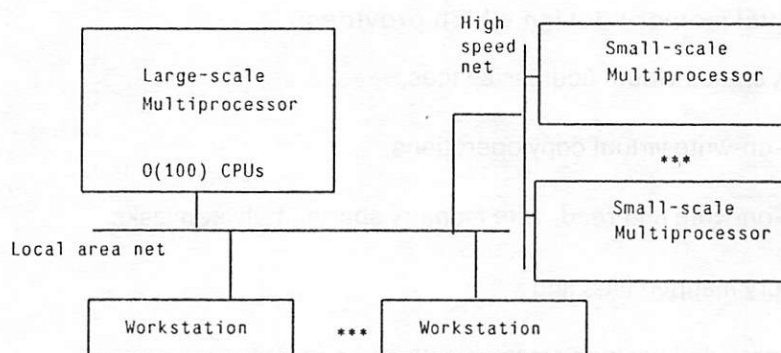


Figure 1-1: The Mach computing environment

2. Design: an extensible kernel

Early in its development, UNIX supported the notion of objects represented as file descriptors with a small set of basic operations on those objects (e.g., read, write and seek) [8]. With pipes serving as a program composition tool, UNIX offered the advantages of simple implementation and extensibility to a variety of problems. Under the weight of changing needs and technology, UNIX has been modified to provide a staggering number of different mechanisms for managing objects and resources. In addition to pipes, UNIX versions now support facilities such as System V streams, 4.2 bsd sockets, pty's, various forms of semaphores, shared memory and a mind-boggling array of ioctl operations on special files and devices. The result has been scores of additional system calls and options with less than uniform access to different resources within a single UNIX system and within a network of UNIX machines.

As the complexity of distributed environments and multiprocessor architectures increase, it becomes increasingly important to return to the original UNIX model of consistent interfaces to system facilities. Moreover, there is a clear need to allow the underlying system to be transparently extended to allow user-state processes to provide services which in the past could only be fully integrated into UNIX by adding code to the operating system kernel.

Mach takes an essentially object-oriented approach to extensibility. It provides a small set of primitive functions designed to allow more complex services and resources to be represented as references to objects. The indirection thus provided allows objects to be arbitrarily placed in the network (either within a multiprocessor or a workstation) without regard to programming details. The Mach kernel abstractions, in effect, provide a base upon which complete system environments may be built. By providing these basic functions in the kernel, it is possible to run varying system configurations on different classes of machines while providing a consistent interface to all resources. The actual system running on any particular machine is a function of its servers rather than its kernel.

The Mach kernel supports four basic abstractions:

1. A *task* is an execution environment in which threads may run. It is the basic unit of resource allocation. A task includes a paged virtual address space and protected access to system resources (such as processors, port capabilities and virtual memory). The UNIX notion of a *process* is, in Mach, represented by a task with a single thread of control.
2. A *thread* is the basic unit of CPU utilization. It is roughly equivalent to an independent program counter operating within a task. All threads within a task

share access to all task resources.

3. A *port* is a communication channel -- logically a queue for messages protected by the kernel. Ports are the reference objects of the Mach design. They are used in much the same way that object references could be used in an object oriented system. *Send* and *Receive* are the fundamental primitive operations on ports.
4. A *message* is a typed collection of data objects used in communication between threads. Messages may be of any size and may contain pointers and typed capabilities for ports.

Operations on objects other than messages are performed by sending messages to ports which are used to represent them. The act of creating a task or thread, for example, returns access rights to the port which represents the new object and which can be used to manipulate it. The Mach kernel acts in that case as a server which implements task and thread objects. It receives incoming messages on task and thread ports and performs the requested operation on the appropriate object. This allows a thread to suspend another thread by sending a suspend message to that thread's *thread port* even if the requesting thread is on another node in a network.

The design of Mach draws heavily on CMU's previous experience with the Accent [7] network operating system, extending that system's facilities into the multiprocessor domain:

- the underlying port mechanism for communication provides support for object-style access to resources and capability based protection as well as network transparency,
- all systems abstractions allow extensibility both to multiprocessors and to networks of uniprocessor or multiprocessor nodes,
- support for parallelism (in the form of tasks with shared memory and threads) allows for a wide range of tightly coupled and loosely coupled multiprocessors and
- access to virtual memory is simple, integrated with message passing, and introduces no arbitrary restrictions on allocation, deallocation and virtual copy operations and yet allows both copy-on-write and read-write sharing.

The Mach abstractions were chosen not only for their simplicity but also for performance reasons. A performance evaluation study done on Accent demonstrated the substantial performance benefits gained by integrating virtual memory management and interprocess communication. Using similar virtual memory and IPC primitives, Accent was able to achieve performance comparable to UNIX systems on equivalent hardware [2].

3. Tasks and Threads

It has been clear for some time that the UNIX process abstraction is insufficient to meet the needs of modern applications. The definition of a UNIX process results in high overhead on the part of the operating system. Typical server applications, which use the fork operation to create a server for each client, tend to use far more system resources than are required. In UNIX this includes process slots, file descriptor slots and page tables. To overcome this problem, many application programmers make use of coroutine packages to manage multiple contexts within a single process (see, for example, [10]).

With the introduction of general purpose shared memory multiprocessors, the problem is intensified due to a need for many processes to implement a single parallel application. On a machine with N processors, for example, an application will need at least N processes to utilize all of the processors. A coroutine package is of no help in this case, as the kernel has no knowledge of such coroutines and can not schedule them.

Mach addresses this problem by dividing the process abstraction into two orthogonal abstractions: the *task* and *thread*. A task is a collection of system resources. These include a virtual address space and a set of port rights. A thread is the basic unit of computation. It is the specification of an execution state within a task. A task is generally a high overhead object (much like a process), whereas a thread is a relatively low overhead object.

To overcome the previously mentioned problems with the process abstraction, Mach allows multiple threads to exist (execute) within a single task. On tightly coupled shared memory multiprocessors, multiple threads within the same task may execute in parallel. Thus, an application can use the full parallelism available, while incurring only a modest overhead on the part of the kernel.

Operations on tasks and threads are invoked by sending a message to a port representing the task or thread. Threads may be created (within a specified task), destroyed, suspended and resumed. The suspend and resume operations, when applied to a task, affect all threads within that task. In addition, tasks may be created (effectively forked), and destroyed.

Tasks are related to one other in a tree structure by task creation operations. Regions of virtual memory may be marked as inheritable read-write, copy-on-write or not at all by future child tasks. A standard UNIX fork operation takes the form of a task with one thread creating a child task with a single thread of control and all memory shared copy-on-write.

Application parallelism in Mach can thus be achieved in any of three ways:

- through the creation of a single task with many threads of control executing in a shared address space, using shared memory for communication and synchronization,
- through the creation of many tasks related by task creation which share restricted regions of memory or
- through the creation of many tasks communicating via messages.

These alternatives reflect as well the different multiprocessor architectures to which Mach is targeted:

- uniform access, shared memory multiprocessors such as the VAX⁴ 11/784, VAX 8300 and Encore MultiMax⁵,
- differential access shared memory machines such as the BBN Butterfly and IBM RP3,
- loosely-coupled networks of computers.

In fact, the Mach abstractions of task, thread and port correspond to the physical realization of many multiprocessors as nodes with shared memory, one or more processors and external communication ports.

4. Virtual Memory Management

The Mach virtual memory design allows tasks to:

- allocate regions of virtual memory,
- deallocate regions of virtual memory,
- set the protections on regions of virtual memory,
- specify the inheritance of regions of virtual memory.

It allows for both copy-on-write and read/write sharing of memory between tasks. Copy-on-write virtual memory often is the result of fork operations or large message transfers. Shared memory is created in a controlled fashion via an inheritance mechanism. Virtual memory related functions, such as pagein and pageout, may be performed by non-kernel tasks. Mach

⁴VAX is trademark of Digital Equipment Corporation.

⁵MultiMax is a trademark of Encore Computer.

does not impose restrictions on what regions may be specified for these operations, except that they be aligned on system page boundaries (where the definition of page size is a boot-time parameter of the system).

The way Mach implements the Unix fork is an example of Mach's virtual memory operations. When a fork operation is invoked, a new (child) address map is created based on the old (parent) address map's inheritance values. Inheritance may be specified as *shared*, *copy* or *none*, and may be specified on a per-page basis. Pages specified as *shared*, are shared for read and write access by both the parent and child address maps. Those pages specified as *copy* are effectively copied in the child map, however, for efficiency, copy-on-write techniques are typically employed. An inheritance specification of *none* signifies that the page is not passed to the child at all. In this case, the child's corresponding address is left unallocated. By default, newly allocated memory is inherited copy-on-write.

Like inheritance, protection may be specified on a per-page basis. For each group of pages there exist two protection values: the current and maximum protection. The current protection controls actual hardware permissions. The maximum protection specifies the maximum value that the current protection may take. The maximum protection may never be raised, it may only be lowered. If the maximum protection is lowered to a level below the current protection, the current protection is also lowered to that level. Either protection is a combination of read, write and execute permissions. Enforcement of these permissions is dependent on hardware support (for example, many machines do not allow for explicit execute permissions, but those that do will be properly enforced).

Consider the following example: Assume that a task with an empty address space has the following operations applied to it:

<u>Operation</u>	<u>Arguments</u>	<u>Comments</u>
allocate	0 - 0x100000	allocate from 0 to 1 megabyte
protect	0 - 0x10000 read/current	make 0 - 64K read only
inherit	0x8000 - 0x20000 share	make 32K - 128K shared on fork

The resulting address map will be a one megabyte address space, with the first 64K read-only and the range from 32K to 128K will be shared by children created with the fork operation.

An important feature of Mach's virtual memory is the ability to handle page faults and page-out data requests outside of the kernel. When virtual memory is created, special paging tasks

may be specified to handle paging requests. For example, to implement a memory mapped file, virtual memory is created with its *pager* specified as the file system. When a page fault occurs, the kernel will translate the fault into a request for data from the file system.

Mach provides some basic paging services inside the kernel. Memory with no pager is automatically zero filled, and page-out is done to a default pager. The current default pager utilizes normal file systems, eliminating the need for separate paging partitions.

5. Virtual Memory Implementation

Given the wide range of virtual memory management built by hardware engineers, it was important to separate machine dependent and machine independent data structures and algorithms in the Mach virtual memory implementation. In addition, the complexity of potential sharing relationships between tasks dictated clean separation between kernel data structures which manage physical resources and those which manage backing store objects.

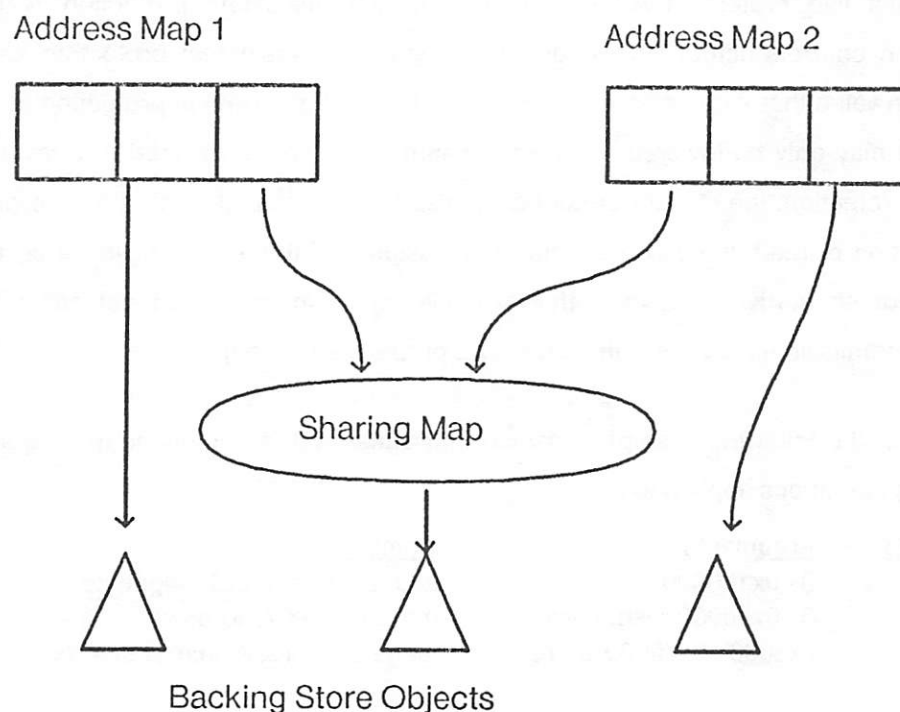


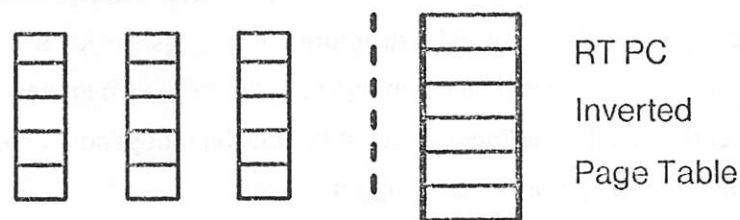
Figure 5-1: Task address maps

The basic data structures used in the virtual memory implementation are:

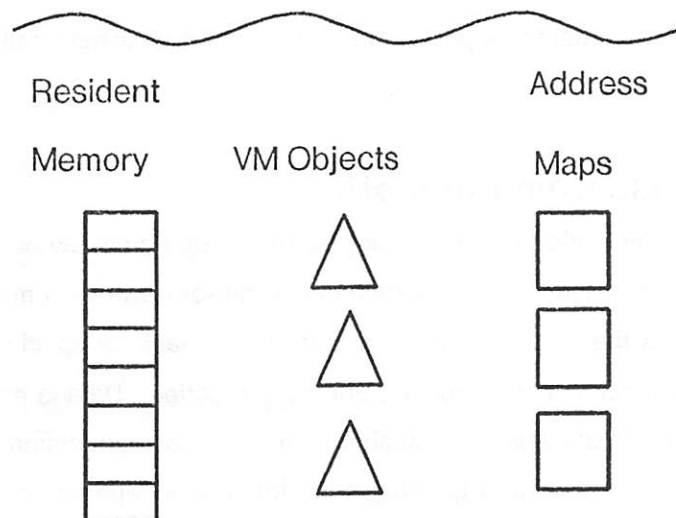
- address maps a doubly linked list of map entries, each describing the properties of a

	region of virtual memory. There is a single address map associated with each task.
share maps	a special address map that describes a region of memory that is shared between tasks. A sharing map provides a level of indirection from address maps, allowing operations that affect shared memory to affect all maps without back pointers.
VM objects	a unit of backing storage. A VM object specifies resident pages as well as where to find non-resident pages. VM objects are pointed at by address maps. <i>Shadow</i> objects are used to hold pages that have been copied after a copy-on-write fault.
page structures	specify the current attributes for physical pages in the system (e.g. mapped in what object, active/reclaimable/free).

Vax Page Tables



Machine Dependent



Machine Independent

Figure 5-2: Task address maps

The virtual memory implementation is split between machine *independent* and machine

dependent sections. The machine independent portion of the implementation has full knowledge of all virtual memory related information. The machine dependent portion, on the other hand, has a simple page validate/invalidate/protect interface, and has no outside knowledge of other machine independent related data structures.

One advantage of this separation is the fact that the "page size" for different sections of the implementation need not be the same. For example, the machine dependent page size on a VAX is 512 bytes. The machine independent page size is a boot time variable that is a power of two of the machine dependent size. The backing storage page size may vary with the backing store object.

The actual data structures used in a machine dependent implementation depends on the target machine. For example, the VAX implementation maintains VAX page tables, whereas the RT/PC implementation maintains an Inverted Page Table. Since the machine independent section maintains all data structures, it is possible for a machine dependent implementation to garbage collect its mappings (e.g. throw away page tables on a VAX). The machine independent section will then request the machine dependent section to map these pages again when the mappings are once again needed.

In addition to the normal demand paging of tasks, the Mach virtual memory implementation allows portions of the kernel to be paged. In particular, address map entries are pageable in the current implementation.

6. Interprocess Communication

Interprocess communication in 4.3 bsd can occur through a variety of mechanisms: pipes, pty's, signals and sockets [6]. The primary mechanism for network communication, internet domain sockets, has the disadvantage of using global machine specific names (IP based addresses) with no location independence and no protection. Data is passed uninterpreted by the kernel as streams of bytes. The Mach interprocess communication facility is defined in terms of *ports* and *messages* and provides both location independence, security and data type tagging.

The *port* is the basic transport abstraction provided by Mach. A port is a protected kernel object into which messages may be placed by tasks and from which messages may be removed. A port is logically a finite length queue of messages sent by a task. Ports may have any number of senders but only one receiver. Access to a port is granted by receiving a

message containing a port capability (to either send or receive).

Ports are used by tasks to represent services or data structures. For example, Flamingo [11], a window manager running under Mach on the MicroVAX II, uses a port to represent a window on a bitmap display. Operations on a window are requested by a client task by sending a message to the port representing that window. The window manager task then receives that message and handles the request. Ports used in this way can be thought of as though they were capabilities to objects in a object oriented system [3]. The act of sending a message (and perhaps receiving a reply) corresponds to a cross-domain procedure call in a capability based system such as Hydra [12] or StarOS [4].

A *message* consists of a fixed length header and a variable size collection of typed data objects. Messages may contain both port capabilities and/or imbedded pointers as long as both are properly typed. A single message may transfer up to the entire address space of a task.

Messages may be sent and received either synchronously or asynchronously. Currently, signals can be used to handle incoming messages outside the flow of control of a normal UNIX style process. A task could create or assign separate threads to handle asynchronous events.

Figure 6-1 shows a typical message interaction. A task A sends a message to a port P2. Task A has send rights to P2 and receive rights to a port P1. At some later time, task B which has receive rights to port P2 receives that message which may in turn contain send rights to port P1 (for the purposes of sending a reply message back to task A). Task B then (optionally) replies by sending a message to P1.

Should port P2 have been full, task A would have had the option at the point of sending the message to: (1) be suspended until the port was no longer full, (2) have the message send operation return a port full error code, or (3) have the kernel accept the message for future transmission to port P2 with the proviso that no further message can be sent by that task to P2 until the kernel sends a message to A telling it the current message has been posted.

Figure 6-2 shows a task A sending a large (for example, 24 megabyte) message to a port P1. At the point the message is posted to P1, the part of A's address space containing the message is marked copy-on-write -- meaning any page referenced for writing will be copied

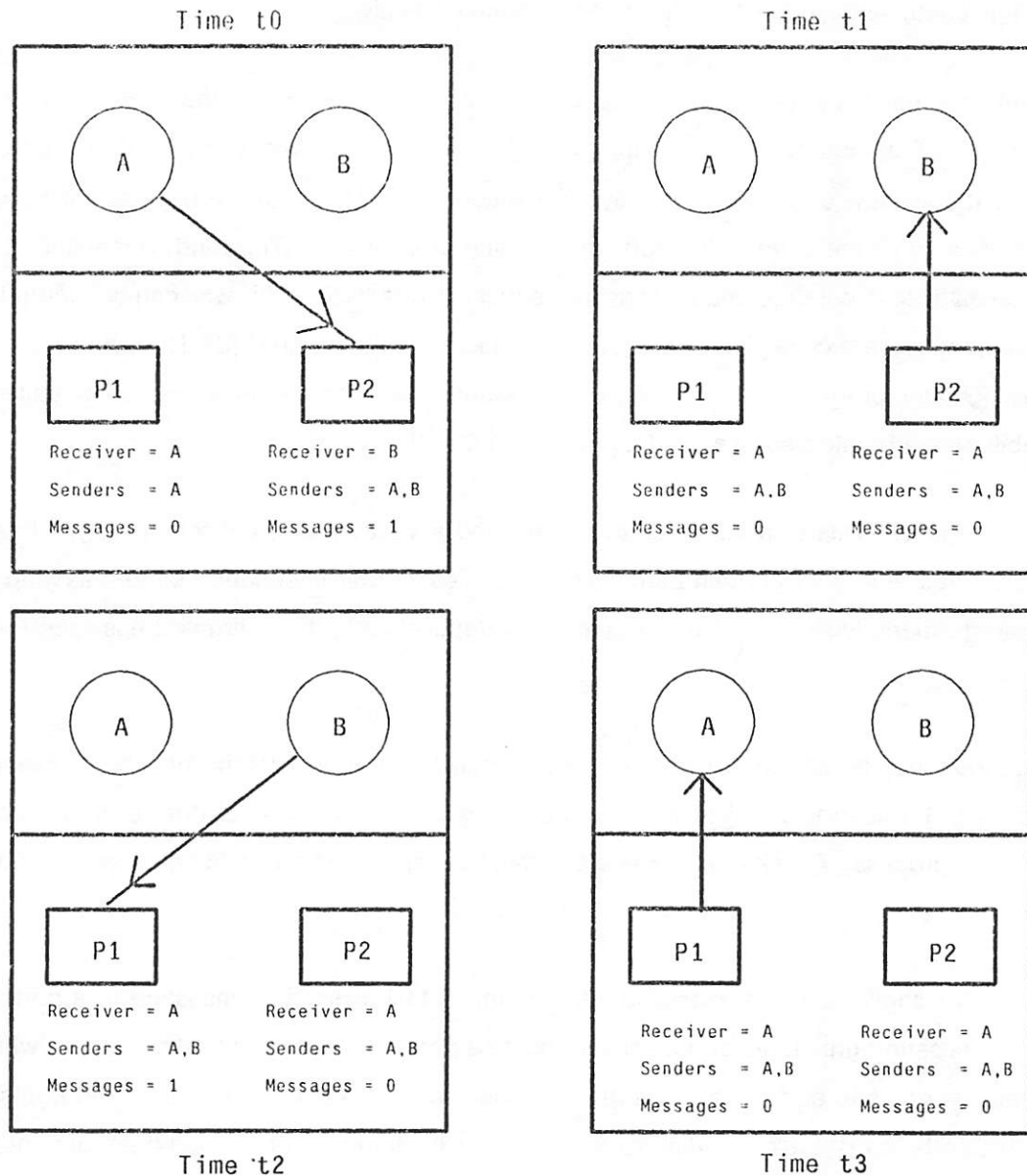


Figure 6-1: Typical message exchange

and the copy placed instead into A's virtual memory table. The copy-on-write data then resides in a temporary kernel address map until task B receives the message. At that point the data is removed from the temporary address map. The operating system kernel determines where in the address space of B the newly received message data is placed, allowing the kernel to minimize memory mapping overhead. Any attempt by either A or B to change a page of this copy-on-write data results in a copy of that page being made and placed into that tasks' address space.

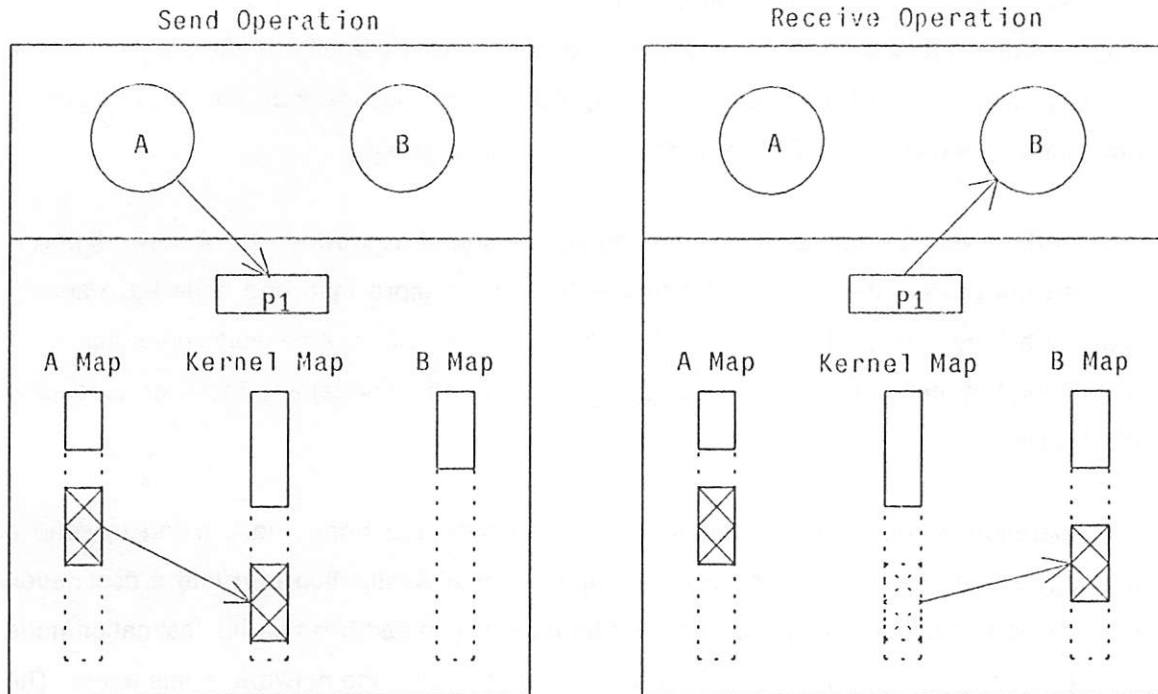


Figure 6-2: Memory mapping operations during message transfer

6.1. Defining interprocess interfaces

Interprocess interfaces, including the interface to the Mach kernel, are defined using an interface definition language called Matchmaker [5]. Matchmaker compiles these interface definitions into remote procedure call stubs for various programming languages including C, CommonLisp and a CMU variant of PASCAL. These stubs use the Mach message system as their basic transport facility. Matchmaker interfaces can perform runtime type-checking and provide sufficient information in messages for network communication servers to perform routine data-type conversion and data re-alignment between machines of different architectural types.

6.2. Network communication and security

By itself, the Mach kernel does not provide any mechanisms to support interprocess communication over the network. However, the definition of Mach IPC allows for communication to be transparently extended by user-level tasks called *Network Servers*. A network server effectively acts as a local representative for tasks on remote nodes. Messages destined for ports with remote receivers are actually sent to the local network server.

When a task sends a message to a destination port on another node, the forwarding of the

message is transparent to the sender. The sender has no direct means of determining whether the eventual destination port is local to its node or is actually on a remote node. The security guarantees of the Mach port capabilities can be extended into the network environment by network servers through the use of encryption [9].

Network servers collectively implement the abstraction of *Network Ports*. A network port is the network representation of a port to which tasks on more than one node have access rights. Each network port is known by its *Network Port Identifier*. A network server maintains a mapping between network ports (accessible to tasks on its node) and their corresponding local ports.

In operation, when a network server receives a message from a task trying to send a message to a remote destination port, it maps the local destination port into a destination network port identifier. The network server then derives the address of the destination node from the network port identifier and sends the message over the network to this node. The destination network server, on receiving the network message, maps the network port identifier into a local destination port and forwards the message to its ultimate destination. Each network server holds receive rights to those network ports for which the receive rights to the corresponding local ports are held by local tasks. Send and ownership rights to network ports are handled in the same way except that send rights to a network port may be held by many network servers. Messages are typed collections of data objects and any message may contain port access rights. Network servers must examine the type tags of data sent or received in messages over the network to recognize the transmission of such access rights and take appropriate action. Currently Mach's network servers handle data-type conversion and re-alignment for three different machine architectures: the DEC VAX, IBM RT PC⁶ and PERQ Systems Corporation PERQ⁷.

7. System Support Facilities

In addition to the basic system support facilities provided by 4.3, Mach provides a kernel debugger and a transparent remote file system.

⁶RT PC is a trademark of International Business Machines.

⁷PERQ is a trademark of PERQ Systems Corporation

7.1. Kernel Debugger

Kernel debugging has always been a tedious undertaking. UNIX systems traditionally have no support for kernel debugging, requiring kernel implementors to "debug with printf's" or other ad hoc methods. The Mach kernel has a built-in kernel debugger (kdb) based on adb⁸. All adb commands are implemented including support for breakpoints, single instruction step, stack tracing and symbol table translation.

In order to aid debugging, as well as study the performance of the kernel, the Mach debugger also supports functions not available in adb. For example:

- enhanced stack traces - stack traces may contain the values of local variables and registers for each stack frame.
- call/return trace support - single stepping may continue without intervention until the next call or return instruction.
- instruction counting - the number of instructions executed between regions of code may be counted.

During the implementation of the system these features have proven invaluable in both debugging and performance tuning.

7.2. Transparent Remote Filesystem

The remote filesystem available in Mach was originally available in 1982 as part of CMU's locally maintained version of 4.1 UNIX. At that time, it supported only a small set of the functions required of a file system: it could read and/or write publicly accessible files. Over the years, the remote filesystem has undergone a steady increase in functionality. Currently, all UNIX functions, such as remote current directories and execution of remote files, are supported.

The remote filesystem is completely transparent to the user. Users may effectively login to a remote filesystem connection to receive all of their normal privileges on the remote filesystem, or they may elect to not login, and receive only "anonymous" access to the remote filesystem.

A small set of kernel hooks redirects remote file operations to remote servers transparently. Each machine wishing to allow remote requests runs a user-mode server process. The kernel sends requests corresponding to operations such as read, write, open and close. The client then performs the appropriate operation, and returns with a reply code and/or data. Data is

⁸This version currently only works on Vaxen.

not cached with one exception: remote execution of files causes a cached copy of the entire file to be read into an inode on a local disk. Subsequent executions of this file cause the kernel to check for a modification of the remote file, if no such modification has been made, then the locally cached copy is executed.

Links to remote filesystems are created using a special file type. While mount points have been used for this purpose in other remote filesystems [1, 10], it was felt that the restriction on the number of mount points (and the need to actually mount such a filesystem) made this option inappropriate. Using special links allows a machine to connect to an arbitrary number of other machines without the need for mounting all possible remote filesystems, and the fear of the mount table overflowing.

8. Implementation: a new foundation for UNIX

The Mach kernel currently supplants most of the basic system interface functions of the UNIX 4.3bsd kernel: trap handling, scheduling, multiprocessor synchronization, virtual memory management and interprocess communication. 4.3bsd functions are provided by kernel-state threads which are scheduled by the Mach kernel and share communication queues with it.

The spectacular growth in size of the Berkeley UNIX kernel over the last few years has made it apparent that continued expansion of UNIX functionality threatens to undercut the advantages of simplicity and modifiability which made UNIX an attractive operating system alternative for research and development. Work is underway to remove non-Mach UNIX functionality from kernel-state and provide these services through user-state tasks. The goal of this effort to "kernelize" UNIX is a substantially less complex and more easily modifiable basic operating system. This system would be better adapted to new uniprocessor and multiprocessor architectures as well as the demands of a large network environment. The success of this transition will depend heavily on the fact that the basic Mach abstractions allow kernel facilities such as memory object management and interprocess communication to be transparently extended. Figure 8-1 shows the eventual relationship between the Mach kernel and UNIX.

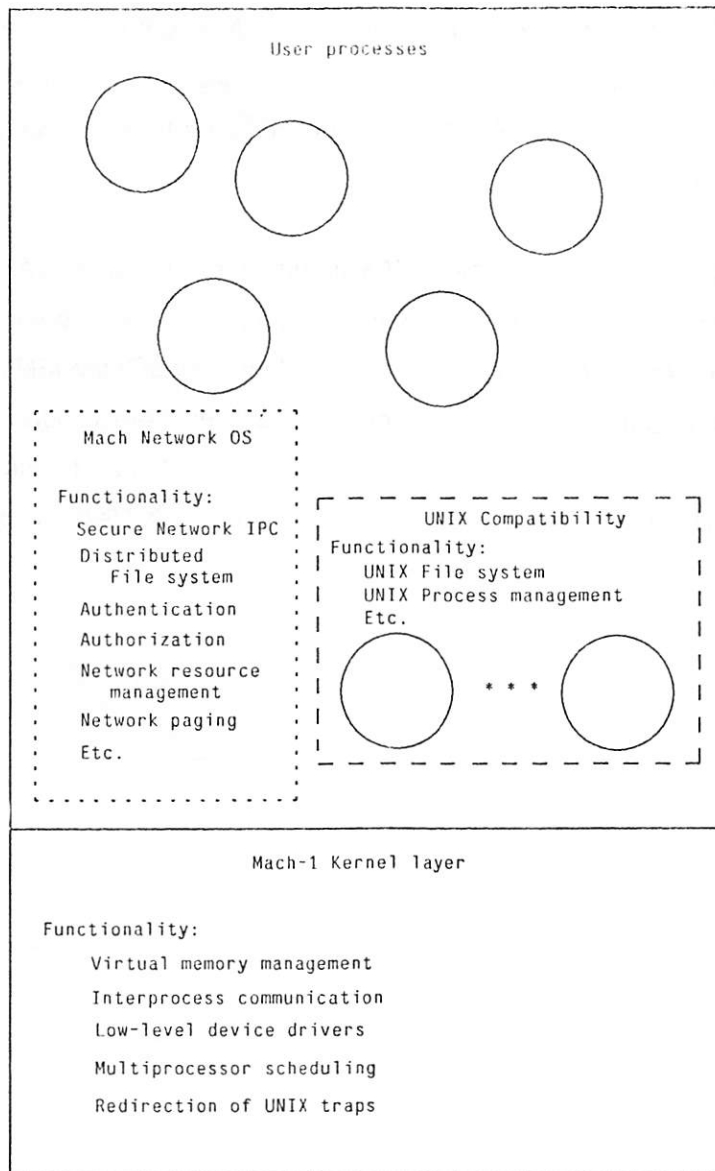


Figure 8-1:
 Mach with UNIX functionality in user-state tasks.
 As of April 1986 the box labeled "UNIX compatibility"
 still executes in kernel state and communicates with the
 Mach kernel layer through a shared communication queue.

9. Current Status: Mach-1

Mach is still under development and extensive performance comparisons with other systems have not yet been done. Although the system has yet to be tuned, current performance appears to be in line with 4.3BSD. Some early simplistic measures of virtual memory performance are encouraging. The MicroVAX II cost of touching newly allocated memory is less than 0.7 milliseconds per 1024 bytes of data (versus approximately 1.2 milliseconds for

4.3BSD). Operations typically expensive in UNIX, e.g. fork, are substantially faster with the new virtual memory support. Mach is currently in production use by CMU researchers on a number of projects including a multiprocessor speech recognition system called Agora and a project to build parallel production systems.

As of April 1986, Mach runs on most VAX architecture machines: VAX 11/750, 11/780, 11/785, 8600, MicroVAX I, and MicroVAX II. In addition, Mach runs on a four (11/780 or 11/785) processor VAX 11/784 with 8 MB of shared memory and the IBM RT PC. The same binary kernel image runs on all VAX uniprocessors and multiprocessors. The same kernel source is used for both VAX and RT PC systems. Work has begun on ports to the uniprocessor SUN 3, multiprocessor Encore MultiMax and VAX 8300. Implementation of the Mach thread mechanism is expected by Summer 1986.

References

- [1] D.R. Brownbridge, L.F. Marshall, B. Randell.
The Newcastle Connection, or UNIXes of the World Unite!
Software - Practice and Experience 12:1147-1162, 1982.
- [2] Fitzgerald, R. and R. F. Rashid.
The integration of Virtual Memory Management and Interprocess Communication in Accent.
ACM Transactions on Computer Systems 4(2):, May, 1986.
- [3] Jones, A.K.
The Object Model: A Conceptual Tool for Structuring Systems.
Operating Systems: An Advanced Course.
Springer-Verlag, 1978, pages 7-16.
- [4] Jones, A.K., R.J. Chansler, I.E. Durham, K. Schwans and S. Vegdahl.
StarOS, a Multiprocessor Operating System for the Support of Task Forces.
In *Proc. 7th Symposium on Operating Systems Principles*, pages 117-129. ACM, December, 1979.
- [5] Jones, M.B., R.F. Rashid and M. Thompson.
MatchMaker: An Interprocess Specification Language.
In *ACM Conference on Principles of Programming Languages*. ACM, January, 1985.
- [6] Joy, W., et. al.
4.2BSD System Manual.
Technical report, Computer Systems Research Group, Computer Science Division,
University of California, Berkeley, July, 1983.
- [7] Rashid, R.F. and G. Robertson.
Accent: A Communication Oriented Network Operating System Kernel.
In *Proc. 8th Symposium on Operating Systems Principles*, pages 64-75. ACM, December, 1981.
- [8] Ritchie, D.M. and K. Thompson.
The Unix Time-Sharing System.
Communications of the ACM 17(7):365-375, July, 1974.
- [9] Sansom, R., Julin, D. and Rashid R.
Extending a Capability Based System into a Network Environment.
Technical report, Department of Computer Science, Carnegie-Mellon University, April, 1986.
- [10] Satyanarayanan, M., et.al.
The ITC Distributed File System: Principles and Design.
In *Proc. 10th Symposium on Operating Systems Principles*, pages 35-50. ACM, December, 1985.
- [11] E. T. Smith and D. B. Anderson.
Flamingo: Object-Oriented Abstractions for User Interface Management.
In *Proceedings of the Winter 1986 USENIX Conference*, pages 72-78. January, 1986.

- [12] Wulf, W.A., R. Levin and S.P. Harbison.
Hydra/C.mmp: An Experimental Computer System.
McGraw-Hill, 1981.

An Extensible I/O System

Jim Rees, Paul H. Levine, Nathaniel Mishkin, Paul J. Leach

apollo computer inc
330 Billerica Road
Chelmsford, MA 01824

Introduction

For years, programming environments have provided **device independent** program I/O. The programmer normally codes file I/O requests using a standard set of procedure calls, such as the Unix *open*, *close*, *read*, and *write* system calls, or language specific I/O calls. This model enables a program written primarily to perform I/O to simple files to also read from keyboards or IPC channels, and to write to display windows or IPC channels without any modification. The intent is to unburden the programmer from the necessity of either binding the program to a specific target for its I/O or enabling the program to adjust to the vagaries of different I/O targets at program run-time. That is, to make the applications program I/O independent of target type.

While this concept has been around for a long time, the systems that implemented the concept have generally had one major shortcoming. The only way to add a new type of I/O target to the system was to modify the system source. In the case of Unix operating systems, for example, it is necessary to modify and rebuild the operating system kernel and to have all of the software that implements the management of the new I/O target permanently wired into physical memory. Most schemes for adding new file types to the Unix kernel operate at the file system level, so that within a given file system, all files have the same type. Further, whenever a new type is added, various pieces of the system have to be modified to behave correctly with respect to the new type. Because of this sizable burden, programmers are discouraged from defining numerous I/O target types.

Our goal was to create a framework in which file I/O could be truly extensible — to allow users to define new types without modification to the basic system. Our work consisted of building a general framework for extensibility and then applying those techniques to stream I/O. We call the framework a **typed object management system**; and the associated file I/O facility **Extensible Streams (ES)**. The combination of these two is called the **DOMAIN® Open Systems Toolkit**.

The system resulting from our work is novel because it:

- Supports (relatively large) typed, permanent, sharable objects in a distributed file system.
- Allows users to define new types of objects.
- Allows users to associate generic procedures (operations) with types; the procedures are dynamically loaded into the address space of processes when the procedure is invoked.

The Open Systems Toolkit allows users to extend the DOMAIN file system by inventing new file types and writing managers for these types. The current implementation allows dynamic creation

Unix is a trademark of AT&T.

DOMAIN is a registered trademark of Apollo Computer Inc.

Copyright 1986 Apollo Computer Inc.

of new types, and dynamic binding of typed objects to the **managers** which implement their behavior. Type managers are written and debugged as user programs and require no kernel modifications for installation. This system has been used successfully to write and debug new device drivers, to add new types of files, and to provide remote file system interconnects to foreign file systems.

DOMAIN Architecture

The DOMAIN system [1] is an architecture for networks of personal workstations and server computers that creates an integrated distributed computing environment. A major component of this distributed system is a distributed file system [2] which consists of four major components: the object storage system, mapped file management, concurrency control and naming service.

The DOMAIN distributed object storage system (OSS) provides location transparent typed object management across a network of loosely coupled machines. We say “object” rather than file to specifically include all of the named non-disk objects in a computing environment, such as devices (serial I/O lines, magtapes, null, etc.), IPC facilities (sockets, etc.) and processes. While a naming service manages a network-wide hierarchical name space, at the OSS level objects are named by a 64-bit **unique identifier** (UID). The UID consists of a time stamp and a unique node ID. This guarantees that the UID is unique across all DOMAIN nodes for all time.

A 64-bit **object type UID** is associated with every object. This type is used to divide the set of all objects into classes of like objects; all of the objects in a class have common properties and must be operated upon by a single set of procedures. We use a UID (rather than any other kind of type identifier) because a system facility supports the unique creation of these 64-bit numbers across all Apollo products. In the basic DOMAIN system there are several types, including ASCII text, binary, directory, and record. This strong typing allows the creator of an object to explicitly specify its intended use and interpretation, rather than depending on the conventions and cooperation of other users and programs.

The DOMAIN OSS supports a consistent set of facilities for naming, locating, creating, deleting, and providing access control and administration over all objects. Each object has an inode, which we have extended to contain (among other things) the type UID of the described object.

For disk-based objects OSS also provides storage containers (arrays of pages) for uninterpreted data. A process accesses this data by handing the kernel the object's UID and asking for it to be **mapped** into its address space. The process then uses ordinary machine instructions to directly manipulate the contents of the object — the single level store (SLS) concept of Multics [3], Pilot [4], and System/38 [5].

Layered on top of the file system is the **Streams** library, a user state library mapped into every process's address space, which provides a traditional I/O environment for programs. The Streams library implements the standard I/O interfaces and so provides equal access to both disk and non-disk resident objects. The DOMAIN Stream operations form a superset of the Unix file I/O operations, as they include record-oriented operations and more inquiry operations but are all based on a file descriptor returned to callers of *open*. Streams is an object-oriented facility in that its behavior is determined by the type of object to which its operations are applied. When a stream operation is invoked, Streams calls the manager that handles operations for the type of the object being operated on. Figure 1 diagrams the relationships among the various pieces of the DOMAIN object management system and Streams.

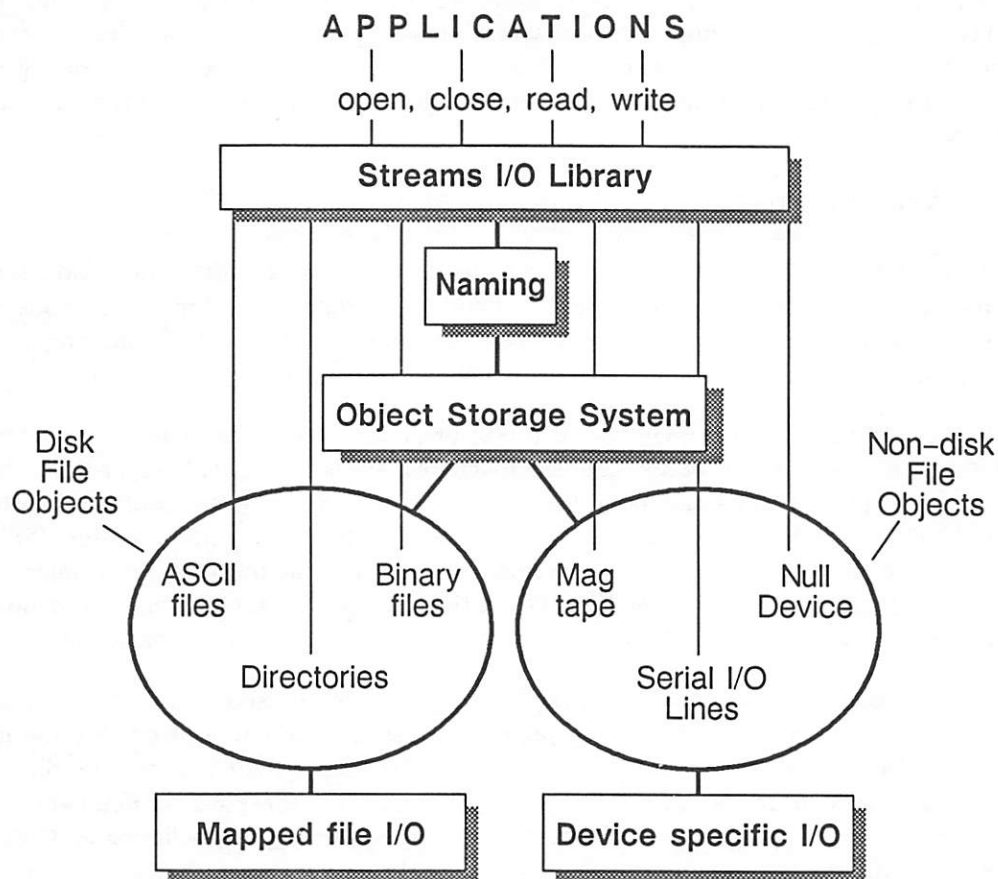


Figure 1. The relationships among the various pieces of the DOMAIN object management system and Streams.

Typed Object Management

The fundamental concept underlying the object-oriented part of our system is the notion that every object is strongly typed and that for each object type there is a set of executable routines that implement a well-specified group of operations on that type. This section describes the object typing strategy, defines operations and describes their partitioning into traits. It also explains the management facilities necessary to associate typed objects with the code that implements the operations defined for them.

Unix file system objects do not have an explicit type tag, but do keep a form of type information in several different places. The *mode* field in a file's inode contains some bits that distinguish among ordinary files, directories, character and block special files (devices), and depending on the version of Unix system, FIFOs, sockets, textual links, and other types of file system objects. There may also be type information coded into the major and minor device numbers to, for example, distinguish between tape drives and disk drives. In some cases, type information is encoded in the first few bits of the file data itself. For instance, there may be "magic numbers" for tagging various flavors of executable (a.out) files.

In the DOMAIN system, the type tag is a UID which is explicitly attached to the object at the time that the object is created. This provides the advantage of a single, common mechanism to

distinguish among all types. The use of a UID (rather than a small integer) allows the arbitrary creation of new types without appealing to a central authority.

The fundamental concept underlying the object-oriented part of our system is the notion of an object type as a set of legal states together with a collection of **operations** that implement the state transitions. Operations can be viewed in two ways: as a specification of how to invoke a transformation on the state of an object, or as the executable code that performs the transformation. The collection of code that implements the set of operations for an object type is known as that object type's **type manager**.

A **trait** is an ordered set of operations. It represents a kind of behavior that a client desires from an object. For example, the operations *open*, *close*, *read* and *write* could be a "stream-like" trait, and the operations *set speed* and *echo input* could be part of a "tty-like" trait. An object supports a trait if its type manager implements the operations of the trait. For every trait that a type manager supports, the manager provides an **entry point vector (EPV)**, that is an ordered list of pointers to the procedures that implement the operations in the trait.

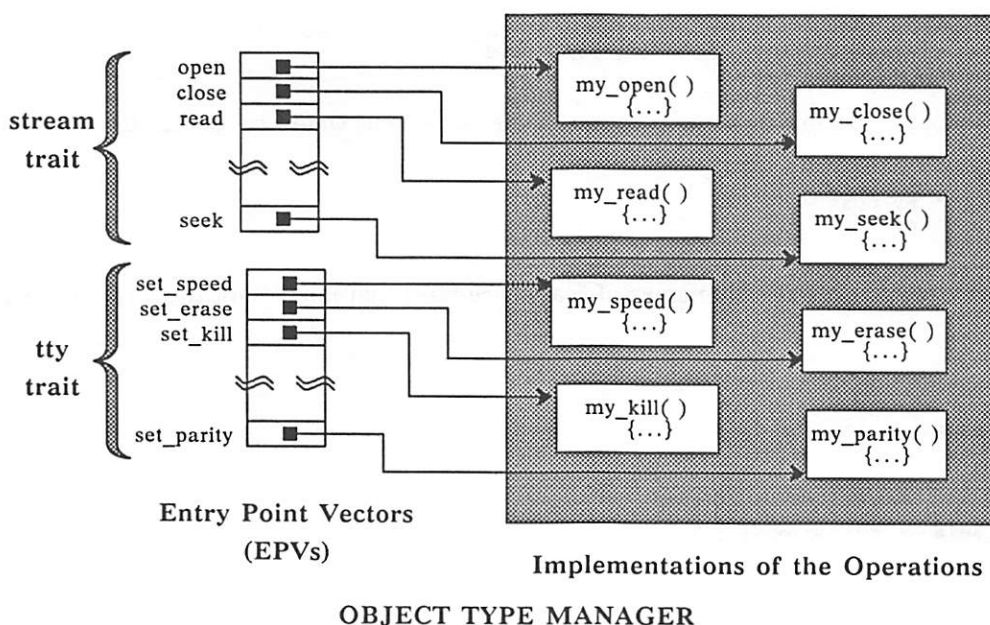


Figure 2. The type manager consists of the routines that implement one or more sets of operations (traits) and the entry point vectors (EPVs) that map the supported operations to the routines that implement them.

The implementation of the typed object management system has two main components: the **type system** and the **trait system**. We use the name **Trait/Types/Managers (TTM)** to refer to these two components plus the set of all type managers.

The type system is responsible for maintaining a data base containing mappings between type UID, type manager, and type name. New types can be created at will. For convenience, there is a name for every type, but a type UID rather than a type name is actually attached to the file system objects. This guarantees that all types are unique, even if two different implementors independently choose the same name. The type system provides procedures that can be used to

create new types, associate a name with a type, and look up type UID of a given type name. It can also find the manager for a given type.

The role of the trait system is to bind <object, trait> pairs to type manager EPVs. It provides the *trait_\$bind* call for this purpose. This call looks up the object's type UID and then asks the type system for the corresponding type manager. Object code libraries containing managers are not pre-linked with client object code. Rather, the trait system is responsible for dynamically loading them into the address space of clients as necessary. To perform this task, the trait manager uses the type system to locate the manager object code file. It then loads the manager into the address space of the client. The type manager is linked as an autonomous program whose main entry point is called when the manager is loaded. The code at this entry point registers all supported <trait, EPV> pairs with the trait system. Once the manager is loaded, the trait system returns the requested EPV to its client.

The type definition for an EPV corresponding to a trait that describes operations on stacks might look like:

```
typedef struct {
    void (*push)(uid_$t, stack_$elem_t);
    void (*pop)(uid_$t);
} stack_$epv;
```

The actual EPV for a type manager that supported the stack trait would be declared as:

```
stack_$epv my_stack_epv = {
    my_push,
    my_pop,
};
```

where *my_push* and *my_pop* are the names of real procedures that implement the *push* and *pop* operations:

```
void my_push(obj, elem)
uid_$t obj;
stack_$elem_t elem;
{
    ...
}

stack_$elem_t my_pop(obj)
uid_$t obj;
{
    ...
}
```

The client uses *trait_\$bind* to get a pointer to an EPV from the trait system:

```
trait_$epv *trait_$bind(obj, trait, typuidp, statusp)

uid_$t obj; /* IN: object we want to operate on */
trait_$t trait; /* IN: trait we want to use */
uid_$t *typuidp; /* OUT: type of object */
status_$t *statusp; /* OUT: status */
```

Once a client has called *trait_\$bind* and received an EPV, it can invoke operations on the object. For example, to call the *push* and *pop* operations in the sample trait above:

```
epv = (stack_$epv *) trait_$bind(my_obj, stack_$trait, &type_uid, &status);
(*epv->push)(my_obj, an_elem);
an_elem = (*epv->pop)(my_obj);
```

The DOMAIN system provides a set of programs for creating and installing new types and their managers. A user who creates a new type will also typically write a type manager for that type. The manager is written as a set of subroutines, each implementing an operation for the traits that the

manager supports. The programmer can use the standard debugging tools on the type manager. The manager is installed by running a program that puts the executable code in a well-known place and registers the new manager with the type system data base. No kernel modifications are required, and the machine does not have to be rebooted. There is no limit on the number of object types a single system may support since their managers are only loaded when needed.

Extensible Streams

Extensible Streams is a client of TTM. ES defines three basic traits: IO, IO_OC, and IO_XOC. The IO trait contains the traditional I/O operations — *get (read)*, *put (write)*, *seek*, etc. The IO_OC trait contains the operations *open* and *initialize*. (The IO_XOC trait is similar to IO_OC except that it supports **extended naming**, a facility that allows non-standard pathnames, described below.) ES also defines a set of **auxiliary traits** that contain operations that only some type managers will choose to implement. The current set of auxiliary traits include: SIO (operations for manipulating serial I/O lines), SOCKET (operations corresponding to the 4.2bsd Unix “socket” system calls), PAD (operations for manipulating windows), and DIRECTORY (operations for reading and manipulating directories).

ES introduces a layer of abstraction on top of the basic operations. This layer — called the **I/O Switch** — supports the notion of an **open stream** and isolates the user of file system I/O from the TTM. An open stream is created by calling the I/O Switch procedure *ios_\$open* which:

- Calls *trait_\$bind* to get the IO and IO_OC EPVs for the object being opened.
- Calls the manager’s *open* operation. This operation returns a **handle** — a virtual address of a descriptor that is meaningful only to the manager. The manager stores in the handle whatever information it needs in order to maintain the semantics of an open stream (e.g., position in stream, buffers).
- Allocates an entry in the **stream table** — a table of open streams. Each entry in this table contains the EPVs for the IO and IO_OC traits, and the handle returned by the *open* operation.
- Returns the small integer — the **file descriptor** — that identifies the table entry allocated in the previous step. This file descriptor is used by the application program on subsequent calls.

Another I/O Switch procedure, *ios_\$create*, is similar to *ios_\$open* except that it creates a new object and calls the manager’s *initialize* operation. In addition to returning a handle, the *initialize* operation stores any information it needs to in the newly-created object.

For each operation in the IO trait, a trivial I/O Switch procedure takes a file descriptor as its first argument, converts the descriptor to a handle (by consulting the stream table), and calls the appropriate procedure from the EPV (also obtained from the stream table). The various forms of I/O (e.g., Unix I/O system calls, FORTRAN and Pascal language I/O primitives) are implemented in terms of these I/O Switch procedures.

Extended Naming

Extended naming is a facility that allows the pathname of an object being opened to be augmented with additional text to be interpreted by the Streams manager of the object to which the pathname refers. This additional text is called the **residual pathname**.

If an application calls the I/O Switch's open procedure with a pathname containing a residual, and the non-residual part of the pathname names an object whose type manager implements the IO_XOC trait (as opposed to the IO_OC trait), then the I/O Switch passes the residual to the manager as one of the arguments in the IO_XOC *open* operation. The manager is free to interpret the residual in any way it chooses.

Program-level I/O based on a simple system naming facility allows an application program to pass the name of a file system object into the *open* call, for the I/O Switch to locate the specified object, and for the manager of that type of object to then do its job. For example, the pathname */usr/fonts/classic* refers to the object whose name is *classic*, which is catalogued in the directory whose name is *fonts*, which in turn is catalogued in a directory object whose name is *usr*. The I/O Switch *resolves* the entire pathname down into the single target object, and passes a shorthand identifier for that object to the manager.

The intent of extended naming is to allow the object managers themselves to take over part of the pathname-walking responsibility so that they can manage a collection of objects that can be distinguished by the remainder of the pathname. To clarify this notion, consider the following.

The pathname */jim/test.c* would normally be interpreted as a file named *test.c* catalogued in the directory named *jim*. The name also suggests that the file is a C language source file and that all operations that would need to work on such a file (e.g. compiling, printing, editing) could be requested by specifying this name.

Now let's suppose that file *test.c* is of type *history*. The actual file system object contains the entire change history of the file, much the same way that a SCCS [7] file does. Programs that do not care about the change history can open this file and read from it. The *open* and *read* requests are passed on by the I/O Switch to the *history* type manager, and the manager can be written so that the program reads the latest version of the file.

Extended naming takes the concept one step further by allowing the manager writer for the *history* object type to allow the specification of additional pathname text. Where the simply specified pathname results in the reading of data from the *latest* version of the file *test.c*, the manager writer might wish to allow a naming syntax of the form */jim/test.c/-1* to indicate that the application wishes to use the penultimate version of the file instead of the newest. The I/O Switch allows this additional specification to be issued at the application program layer and passed through to the manager for the target object.

The application passes the pathname (with the extended name) to the I/O Switch *open* routine. The *open* routine evaluates the pathname one pathname component at a time walking from left to right. In the current example, *jim* is a directory where the name *test.c* is located. *test.c* is discovered to be a *history* file (not a directory), and because the original pathname still has remaining text ('-1') that the I/O Switch cannot resolve, it passes that remainder to the *history* object manager's IO_XOC *open* routine. The *history* manager is then able to decide what text to provide to subsequent *read* requests and the intended result occurs. In this case, the application program is not affected by the apparent peculiarity of the original pathname. The I/O Switch avoids confusion by only walking the pathname through objects that support the *directory* trait and the manager is able to get whatever information it needs to do the job it was written to do.

Other examples of extended names a *history* manager might be willing to accept are:

```
/jim/test.c/03.02.85  
/jim/test.c/original  
/jim/test.c/yesterday
```

Another example of the application of extended naming is a gateway to a non-DOMAIN file system. For example, imagine an object whose name is *THEM* and whose type is *UNIX_gate*. A pathname of the form */gateways/THEM/usr/lan/test.c* could be passed by an application program to the I/O Switch. The Switch would see that the object whose name was *gateways* was a directory and would look the name *THEM* up in that directory. *THEM* would be found to be a *UNIX_gate* object, and since the Switch cannot walk the pathname through objects that are not directories, it would call the *UNIX_gate* object manager's *open* routine. That routine is passed the UID for the object whose name is *THEM* and the remaining pathname (*/usr/lan/test.c*). The *UNIX_gate* manager then has the information it needs to contact a remote file service for the data it needs to meet the demands of the requesting application program. The protocol that the manager uses to access the remote files is entirely up to the manager writer, and because the manager runs in user space, it is not restricted to kernel services but can use any service available at the user level. This scheme has been used to build a type manager that interconnects the DOMAIN file system with a generic Berkeley 4.2 Unix file system, and another that connects to a VAX/VMS file system.

Underlying Facilities

Many facilities provided in the DOMAIN environment made the implementation of TTM and Extensible Streams possible. These facilities make it possible to write OS-like functions in user space.

The underlying virtual memory system — which allows objects to be mapped into the virtual address space — is needed to give type managers low level, yet controlled, access to the raw data in objects. The virtual memory system allows more flexible access to the address space than that allowed by *sbrk(2)*. These calls take the name of an object, map the object into the address space, and return a pointer to (i.e. the virtual address of) the mapped object. The address space of a process can be characterized solely in terms of what objects are mapped where. Processes are not allowed to make memory references to parts of the address space to which no object is mapped.

The read/write storage (RWS) facility is a flexible and efficient storage allocation mechanism. It is implemented in user space in terms of the virtual memory primitives; it maps temporary objects into the address space and allocates storage from that part of the address space. It allows storage to be allocated from multiple pools. One pool corresponds exactly to the type of storage allocated by *malloc*. Another pool is similar, except its state is not obliterated by *exec* calls. Type managers must use storage from this pool to hold per-process state information since open streams must survive calls to *exec*.

RWS also provides a global storage pool. The global pool is a place where storage that can be viewed from all processes' address spaces can be allocated. The allocation call returns a pointer to the allocated storage, and this pointer is valid in all processes. Type managers must use storage from the global pool to maintain things like the current position (i.e. offset from beginning-of-file) of an open stream. If a process opens a stream to an object, forks, and then the child does I/O to the stream it was passed, the parent sees the position of *its* stream change too. Thus, position information must be in storage accessible to both parent and child. Because type managers run in user space, they need a user space global storage allocator for this purpose.

The dynamic program loader allows the system to load managers as they are needed. Managers for types that are not used by a given process do not take up any virtual address space in that process. The loader is implemented in user space in terms of the RWS facility (to allocate space for static data) and the mapping calls. The pure parts of executable images are simply mapped into the address space before execution, because the compilers produce position-independent code. In 4.2bsd, only the kernel can be dynamically linked to; all other subroutines must be statically bound to the program which uses them.

The eventcount [8] (EC2) facility is the basic process synchronization mechanism. Eventcounts are similar to semaphores: eventcounts are associated with significant events, and processes can advance an eventcount to notify another process that an event has occurred, or wait on a list of eventcounts until the first event happens.

A design principle for all DOMAIN interfaces is that for every potentially blocking procedure in an interface, there is an associated eventcount that can be obtained through the interface and that is advanced when the blocking procedure would have unblocked. This always allows programs to wait for multiple events (say, input on a TTY line and arrival of a network message) simultaneously. The 4.2bsd *select(2)* system call is implemented in terms of eventcounts. However, unlike *select*, eventcounts can also be used to wait on non-I/O events, such as process death.

The mutual exclusion (MUTEX) facility is a user-state library that contains calls that allow multiple processes to synchronize their access to shared data (i.e. data in objects that are mapped into multiple processes). MUTEX is implemented in terms of EC2. MUTEX defines a lock record that consists of a lock byte and an eventcount. Typically, applications embed a record of this type in a data structure over which mutual exclusion must be maintained. A MUTEX lock is set by calling *mutex_\$lock*, which attempts to set the lock byte (using the hardware test-and-set instruction). If it fails to set the lock byte, it waits on the eventcount; when the wait returns, *mutex_\$lock* repeats the attempt to set the lock byte. *mutex_\$unlock* unlocks a MUTEX lock by clearing the lock byte and advancing the eventcount. Type managers use shared storage to maintain various kinds of information. To control access to this data, managers use the MUTEX facility.

The shared file control block (SFCB) facility allows multiple processes to coordinate their access to the same object. There is various dynamic information that processes might want to keep about an object. For example, type managers need to maintain information about the object's current length, whether the object is being accessed for read or write, and whether other processes should be allowed to concurrently access the object. Since this information must be accessed by multiple processes, it must reside in global storage. The first process to access the object can allocate the storage, but how are other processes to find the virtual address of that storage? The SFCB facility addresses this problem by maintaining a table translating object UID into global virtual address. (The table is in global storage at a well-known location.) The *sfcbl_\$get* call takes an object UID and returns a pointer to a piece of global storage (called the SFCB). If no storage was "registered" with SFCB prior to the call, an SFCB is allocated and registered under the specified UID; otherwise, a pointer to the existing storage associated with that UID is returned and a use count field in the storage is incremented to reflect the additional "user" of the storage. *sfcbl_\$free* decrements the use count and, if it reaches zero, frees the storage.

Examples

Extensible Streams allows a number of special-purpose types to be defined. For example:

- History objects: objects that contain many logical versions, only one of which is presented through the open stream at a time. The residual text is used to specify a particular version; if omitted, the most recent version is presented. Useful for source control systems.
- Circular objects: objects that grow to a certain size and then have their “oldest” data discarded when more data is written to them. Useful for maintaining bounded log output from long-running programs.
- Structured documents: objects that contain document control (e.g. font and sectioning) information but which can be read through an open stream as if they were simple ASCII text. Useful for using conventional text processing tools (e.g., Unix “grep”) [9].
- Gateways to non-DOMAIN file systems: objects that are placeholders for entire remote file systems. The residual is used to specify a particular file on the remote system. The manager implements whatever network protocol it chooses to access the remote system's data.
- Distributed, replicated data bases: objects that, for reliability reasons, are distributed across a network of machines. A Yellow Pages [10] manager would eliminate the need for the *ypcat* command, and allow any ordinary user to access a Yellow Pages data base without modification and without having to bind to a special library (the type manager, in effect, is the library).

TTM can be used independently of Extensible Streams. For example, the DOMAIN graphics library may be converted to use TTM. Currently, the graphics library has code for all the display hardware types it must support. A TTM-based implementation would define multiple types, one for each type of display hardware, a trait that contains graphics operations (e.g. *move*, *draw*, *trapezoid_fill*), and a set of managers, one per type. This approach would make it possible for only the code necessary for a particular display hardware type to be loaded into the system, and for the graphics library to be easily extensible to new hardware types.

Experience

While the original Streams library was written with the idea of types and type managers in mind, the actual implementation had to be restructured substantially to take advantage of TTM. We took this opportunity to redesign the interface to managers and the interface presented to applications that use the Streams library.

The decision to implement the Berkeley socket calls in terms of a trait turned out to be a good one. On a standard Berkeley Unix system, defining and implementing a new domain (address family) is a fairly difficult task — it requires working inside the kernel. With Extensible Streams, you need only create a new type and implement the `SOCKET` trait in the manager for that type. We have already implemented a manager for “DOMAIN domain sockets”. Currently, this domain supports only datagram-oriented sockets (`SOCK_DGRAM`) because our short-term goal was merely to allow access to specific, low-level DOMAIN networking primitives using the generic, high-level socket calls.

The nature of the address family space made our task a bit more complicated. Address families are identified by small integers in a space over which there is no central authority. As a result, one has to simply pick an address family out of thin air and hope no one else has picked it too. It is

interesting to contrast this state of affairs with the type UID approach we took in TTM, since the small integer address families are essentially type tags. The type UID approach does not have the problem of more than one person picking the same type tag. We did not have the option to change the way address families are identified, so we used a scheme in which address families are translated into type UIDs.

The socket creation primitive is called *socket_\$create_type*. This calls takes a type UID (and a socket type) and returns a stream to a socket of that type. (*socket_\$create_type* is analogous to *ios_\$create* except that it calls the *create* operation in the SOCKET trait instead of the *initialize* operation in the IO_OC trait.) The *socket* system call converts its address family argument into a type UID by consulting an object in the file system that contains a table translating address families into type UID. It then calls *socket_\$create_type*. Note that we could have simply hardcoded a “switch” statement on address family into the implementation of *socket*, but this would have meant that *socket* would not have been as extensible as we would like. (User-defined sockets could have been created via *socket_\$create_type*, but not by *socket*). The scheme we implemented is less than ideal in that it requires both that the type be created and that the address-family-to-type-UID object be updated, but it was the best we could do.

One difficult problem that we have not adequately addressed is that of expanding wildcards in an extended name. For example, using our VMS gateway type manager, one would like to type the name:

```
/gateways/my_vms_sys/dra0:[rees.*]mail.txt
```

If *my_vms_sys* is a gateway object to a VMS system, and *dra0:[rees.*]mail.txt* is a VMS file specification, this specification should be expanded to include files named *mail.txt* in all subdirectories of *dra0:[rees]*. Unfortunately, the agent doing the wildcard expansion (typically the Unix shell) has no knowledge of the syntax of the extended part of the name, and so has no way to expand the wildcard. We considered implementing a “wildcard trait,” but this is difficult to specify in a general way, and every program that does wildcard expansion would have to be modified to use this trait. Instead, we require that standard Unix hierarchical names with “/” separators be used whenever wildcards are being expanded, but we also allow non-standard syntax (as in the example above) if there are no wildcards.

The semantics of certain Unix operations turned out to be fairly obscure. For example, suppose a program sets the FAPPEND flag (via *fcntl(2)*) to “true”, then forks, then the child sets the flag to “false”. Is the change to the stream state seen by the parent as well? We were frequently obliged to look at Unix kernel source or to write sample programs and run them on a standard Unix system to answer our questions. As we discuss below, we are led to believe that the task of producing exact semantic specification is a forbidding one. The various Unix standards committees have their work cut out for them if they intend to do a complete job.

Another interesting experience gained during the implementation of TTM and Extensible Streams relates to the problem of documentation. The goal of Extensible Streams is to make it possible for people who are not employees of Apollo Computer to write new type managers without having access to Apollo source code. This means that the specification of the semantics of the operations must be very precise — it must completely characterize the expectations of application programs that do I/O. The creation of this specification turned out to be a non-trivial task.

Acknowledgements

In addition to the authors, James Hamilton, David Jabs, and Eric Shienbrood worked on the implementation of TTM and Extensible Streams. John Yates was involved in some of the early design work. Elizabeth O'Connell wrote most of the documentation.

References

- [1] Paul J. Leach, Paul H. Levine, Bryan P. Douros, James A. Hamilton, David L. Nelson, Bernard L. Stumpf, "The Architecture of an Integrated Local Network," *IEEE Journal on Selected Areas in Communication*, SAC-1, 5 (November 1983).
- [2] Paul J. Leach, Paul H. Levine, James A. Hamilton, Bernard L. Stumpf, "The File System of an Integrated Local Network," *Proceedings of the ACM Computer Science Conference*, New Orleans, La. (March 1985).
- [3] E. I. Organick, *The Multics System: An Examination of Its Structure*, M.I.T. Press (1972).
- [4] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, S. C. Purcell, "Pilot: An Operating System for a Personal Computer," *Communications of the ACM*, 23, 2 (February 1980).
- [5] R. E. French, R. W. Collins, L. W. Loen, "System/38 Machine Storage Management," *IBM System/38 Technical Developments*, IBM General Systems Division (1978).
- [6] Paul J. Leach, Bernard L. Stumpf, James A. Hamilton, Paul H. Levine, "UIDs as Internal Names in a Distributed File System," *Proceedings of the 1st Symposium on Principles of Distributed Computing*, Ottawa, Canada (August 1982).
- [7] M. J. Rochkind, "The Source Code Control System," *IEEE Transactions on Software Engineering* (December 1975).
- [8] David P. Reed and Rajendra K. Kanodia, "Synchronization with Eventcounts and Sequencers," *Communications of the ACM* (February 1979).
- [9] J. Waldo, "Modelling Text as a Hierarchical Object," *Usenix Conference Proceedings*, Atlanta, Ga. (June 1986).
- [10] B. Lyon and G. Sager, "Overview of the Sun Network File System," Sun Microsystems, Inc. (January 1985).

PATHALIAS
or
The Care and Feeding of Relative Addresses

Peter Honeyman

Computer Science Department
Princeton University
Princeton, New Jersey 08544
princeton!honey

*Steven M. Bellovin**

AT&T Bell Laboratories
Murray Hill, New Jersey 07974
ulysses!smb

ABSTRACT

Pathalias computes electronic mail routes in environments that mix explicit and implicit routing, as well as syntax styles. We describe the history of *pathalias*, its algorithms and data structures, and our design decisions and compromises.

Pathalias is guided by a simple philosophy: get the mail through, reliably and efficiently. We discuss the principles of routing in heterogeneous environments necessary to make this philosophy a reality.

HISTORY AND OVERVIEW

UUCP,¹ the basic networking component of UNIX,² is the backbone of a widespread store-and-forward network. Because setting up new connections is easy, and does not require the intervention of a central administrator, the network has no regular topology. Mail routing is explicitly specified by users. That is, a user who wishes to send mail to `hostb` using `hosta` as a relay would write

`mail hosta!hostb!user`

When the UUCP network was small and the average connectivity was high, explicit routing was a minor annoyance at worst. Most paths were direct, and only a tiny fraction involved more than one or two hops, so remembering proper paths was easy.

Then came USENET.³ For several reasons, UUCP routes soon became a major headache. First, many of the universities on USENET had a low degree of connectivity to other UNIX sites,

* Much of the work was performed at the Department of Computer Science, University of North Carolina at Chapel Hill.

¹ D.A. Nowitz and M.E. Lesk, "A Dial-Up Network of UNIX Systems," in *UNIX Programmer's Manual*, Seventh Ed., 1979.

² UNIX is a trademark of AT&T Bell Laboratories.

³ S.M. Bellovin and M. Horton, "USENET — A Distributed, Decentralized News System," unpublished manuscript, 1986.

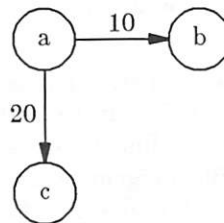
typically with only two or three long-distance links. Second, USENET readers tended to reply along the USENET paths; these were rarely optimal, and were sometimes unusable. Third, as other networks were used for USENET transport, mail reply syntax became complicated by the variety of standards in use.

As USENET grew, it became clear that the UUCP network needed a routing tool, one that took as input a network connectivity graph and generated usable paths to every known destination. *Pathalias* is such a tool. Given a description of the connectivity of the UUCP network, it produces a "least cost" path to every known site.

Of course, any such effort relies heavily on the quality of the connectivity data. At first, gathering such data was a difficult administrative problem. Very few system administrators were willing to spend time compiling lists of neighbors and associated cost data. Some connections could be inferred from USENET maps, but these data were unreliable and lacked cost estimates. Worse, they tended to understate the connectivity of the network, putting more load on coöperative sites. Because the data were often contradictory and error-filled, it was necessary to inspect and edit the data manually. Thanks to the USENIX Association's UUCP-mapping project,⁴ the picture is much brighter today, with timely and accurate data widely available on USENET.

INPUT

The input to *pathalias* is a description of a directed graph that models the connection topology of electronic networks. Each edge is assigned a non-negative cost value and a "routing operator"; the latter shows what character is used for mail-routing, and whether it appears to the right or left of the destination host. For example, the graph



is described as follows, assuming that host *a* uses the UUCP syntax convention of *host!user* for network mail:

```
a      b(10) , c(20)
```

If host *a* uses the ARPANET syntax of *user@host*, the description is

```
a      @b(10) , @c(20)
```

The @ itself indicates the character used for building an address; its position indicates that the host name is on the right. Thus, the default case may be written explicitly:

```
a      b!(10) , c!(20)
```

Many sets of hosts are fully connected, *i.e.*, every host in the set talks to every other. To avoid the necessity of explicitly listing every connection, *pathalias* supports a network notation. Thus,

⁴ M.R. Horton, K. Summers-Horton, and B. Kercheval, "Proposal for a UUCP/USENET Registry Host," in *Proc. Summer USENIX Conference*, Salt Lake City, 1984.

```
dopey    grumpy(10), sleepy(10)
grumpy   dopey(10), sleepy(10)
sleepy   grumpy(10), dopey(10)
```

can be written as

```
UNC-dwarf = {dopey, grumpy, sleepy}(10)
```

where UNC-dwarf is the name given to that network.

As is common on UNIX, data for *pathalias* may be read from the standard input or from a list of input files. Typically, each input file represents the data for a given machine or site; file boundaries have semantic implications in the treatment of “private” names and in resolving duplicate connection data.

The definition of a satisfactory cost metric proves troublesome. Possible metrics include the actual telephone cost of a connection, the nominal frequency of contact (many academic sites are passive — rather than calling out, they wait for other sites to call them), or the transmission speed of a connection. We adopt pragmatic approach: given a choice of paths, we attempt to choose the one that experienced users prefer, as exemplified by existing network traffic. The cost measure is tuned so that *pathalias* produces routes agreeing with these choices.

Using this metric helps balance conflicting concerns. For example, long distance costs often matter less to large corporations than to universities. On the other hand, sites with autodialers have far more freedom to connect with whom they wish. A pragmatic metric also accounts for differences in the reliability of sites; UUCP was not always as reliable as it is today. Actual transmission speed is less important than one might assume; call setup time and the time between calls tend to be the dominant factors, at least for mail messages.

With the basis for a metric in hand, symbolic names like HOURLY, DAILY, *etc.* are assigned numeric values. Early on, these numbers were juggled until, in the estimation of experienced users, the paths produced were reasonable.

Dial-up service is specified as DEMAND for a site that is called whenever there is traffic, or its high-grade kin DIRECT, for local phone calls. In the early days, the odds of completing a call were disappointingly low; port contention, line noise, and difficulties with UNIX’s baud-rate switching were common problems. (This judgement may have been unduly colored by local experience. Several important sites had few dial-in lines, or were served by antiquated telephone switching equipment.) DEDICATED connections — two machines hard-wired together — are considered much higher-grade. A complete table is shown below.

<i>Symbol</i>	<i>Value</i>
LOCAL	25
DEDICATED	95
DIRECT	200
DEMAND	300
HOURLY	500
EVENING	1800
POLLED	5000
DAILY	5000
WEEKLY	30000

Costs can be expressed as arbitrary arithmetic expressions, mixing numbers and symbolic values. For example, HOURLY*3 describes a connection that is completed once every three hours.

In theory, factors that influence cost are additive; in practice, experience shows that the per-hop overhead in time and reliability is so high that it is important to keep paths short. Thus, for example, DAILY is 10 times greater than HOURLY, instead of 24.

OUTPUT

Although it would be convenient to compute the path to a destination as needed, the cost of the calculation is prohibitively expensive. Consequently, *pathalias* precomputes paths to all destinations listed in the input data; these paths are retrieved as needed. The string `%s` is included in output paths as a marker to indicate where the user name should be inserted. Use of such a marker enables the generated path to be used directly as a format string for *printf*. Consider the following input data (a simplified portion of the map from 1981):

```
unc      duke (HOURLY) , phs (HOURLY*4)
duke     unc (DEMAND) , research (DAILY/2) , phs (DEMAND)
phs      unc (HOURLY*4) , duke (HOURLY)
research duke (DEMAND) , ucbvax (DEMAND)
ucbvax   research (DAILY)
ARPA     = @{mit-ai, ucbvax, stanford} (DEDICATED)
```

If run from `unc`, the following output is produced by *pathalias*:

```
0      unc      %s
500    duke     duke!%s
800    phs      duke!phs!%s
3000   research duke!research!%s
3300   ucbvax   duke!research!ucbvax!%s
3395   mit-ai   duke!research!ucbvax!%s@mit-ai
3395   stanford duke!research!ucbvax!%s@stanford
```

There are several points worth noting about the output. First, all generated paths route mail through `duke`, despite the presence of a direct connection to `phs` from `unc`. The reason for this is obvious, given the cost difference on the links to `duke` and `phs`.

Second, mail to ARPANET sites employs mixed-syntax addressing; the path to `ucbvax` uses UUCP conventions (i.e., the host name on the left, delimited by an '!'), while the ARPANET portion has the host name on the right, delimited by an '@'.

Finally, output from *pathalias* is a simple linear file, in the UNIX tradition. If desired, a separate program may be used to convert this file into a format appropriate for rapid database retrieval.

DATA STRUCTURES

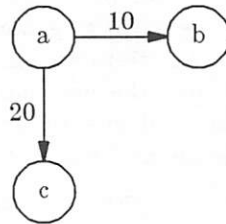
Pathalias runs in three phases: parse the input, build a shortest path tree, and print the routes. Each phase manipulates an in-memory representation of a directed graph. Before describing these computations, we describe the basic data structures.

Graph representation

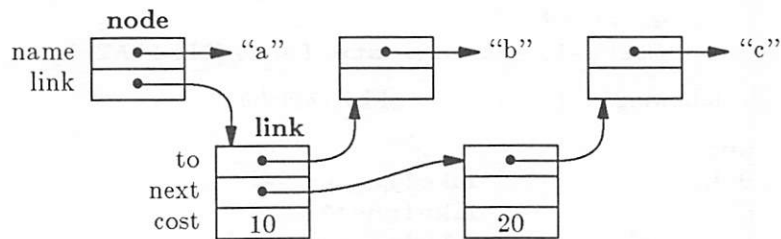
We assume that the world can be modeled as a set of hosts and networks, called *nodes*, with communication links among them. This in turn is modeled by a directed graph, with vertices representing hosts and networks, and edges representing communication links. Edges are weighted with a non-negative cost, and labeled with information describing the syntax used in building addresses from a host to its neighbor. In the input phase, *pathalias* builds an adjacency list representation of the host connectivity graph.

A node is represented by a structure consisting mostly of pointers and flags. One of the fields in a node is a pointer to a singly-linked list of adjacent hosts. A list element, called a *link*, contains a pointer to the next link on the list, a pointer to the destination host on the edge it represents, a non-negative cost, and some flags.

Thus we represent the graph



with these data structures



Networks

An accurate model must take into account various networks, ranging from the ARPANET to local Ethernets. A network has the property that its member hosts share a common name space. We model this with a fully connected graph, or *clique*.

A clique with n vertices contains about n^2 edges, so with over 2,000 hosts in the ARPANET we are faced with millions of edges. To avoid a quadratic explosion in time and space complexity, we represent a network as a single node, with a pair of edges between the network and each member. The following figure depicts this representation (on an undirected graph).



While network declarations also show edge weights and labels, the weight applies only to the edges originating at network members; the weight of edges from the network node to its members is zero. As an analogy, consider automobile toll collection by the Port Authority of New York and New Jersey: you pay to get into the City, but you get back to Jersey for free. Here, you pay to get onto a network, but you get off for free. This preserves the cost structure of the clique representation.

PARSING

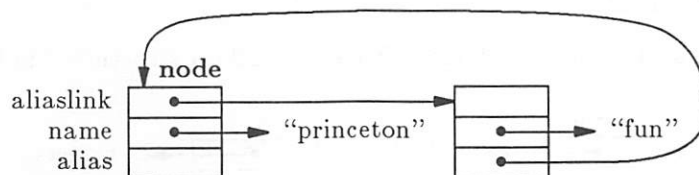
Parsing is done with *yacc*.⁵ We use syntax-directed translation to support a rich syntax with edge weights and labels, aliases, networks, and accommodation of host name collisions. We

⁵ S.C. Johnson, "YACC — Yet Another Compiler Compiler," in *UNIX Programmer's Manual*, Seventh Ed., 1979.

experimented with *lex*⁶ for transforming the raw input into lexical tokens, but were disappointed with its performance: half the run time was spent in the scanner. Since our input tokens are easy to recognize, we built a simple scanner and cut the overall run time by 40%.

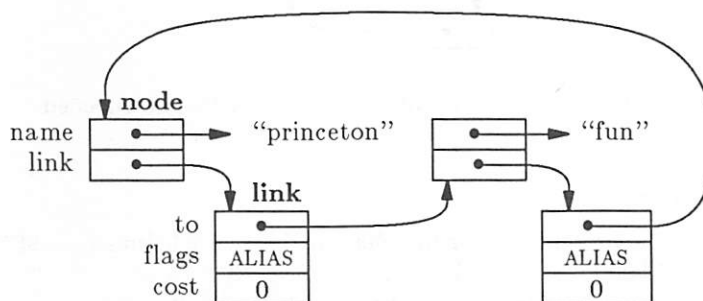
Aliases

Host name aliases are useful when a host is known by several names, or when a host name changes. Originally, aliases were used to assign a set of nicknames to a host. One name, called the *primary* host name, was used in routes; the other names were synonyms for the primary name. Under this model, the graph representation requires two pointers in each node structure: one to link all the aliases of a node onto a list, and one to show the primary host name for the node. For example, a host `princeton` with nickname `fun` has this graph representation.



This treatment does not adequately address the problem of hosts with different names on different networks. For example, the ARPANET host `nosc` has UUCP name `noscvox`. A route by way of the ARPANET must use the former, while a route by way of UUCP must use the latter. Selecting the primary host name requires predicting the shortest path *a priori*.

To address this problem, we discard the notion of a primary host name and treat all aliases as equal. A pair of zero cost edges connects aliases, giving the following as the representation of the above example:



The critical point is that aliases are a property of edges, not vertices. When a host is known by several names, this alias mechanism assures that the name used in a path is the one understood to a host's predecessor.

Host name collisions

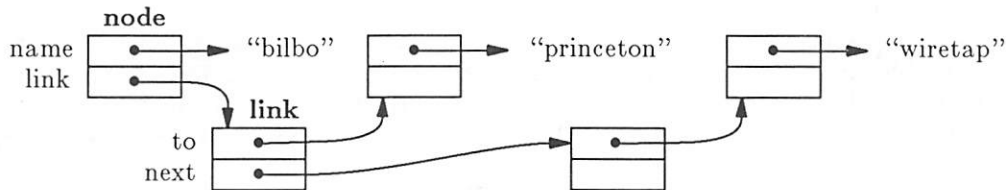
Among the hosts that rely on UUCP as their only means of communication, there is no way to prevent administrators from choosing a host name already in use, and such conflicts do occur. Consequently, *pathalias* may mistakenly merge connection information for distinct hosts with identical names.

We address this difficulty with the "private" declaration, which narrows the scope of connection declarations. Normally, the scope of a host name is global, ranging across all input files. The

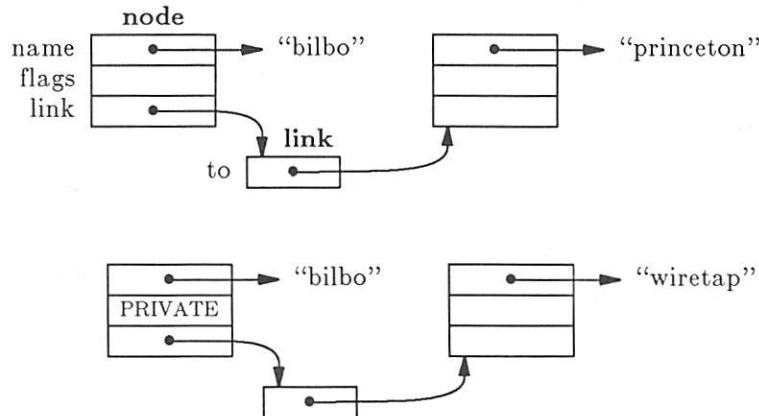
⁶ M.E. Lesk and E. Schmidt, "LEX — Lexical Analyzer Generator," in *UNIX Programmer's Manual*, Seventh Ed., 1979.

scope of a private declaration extends to the end of the file in which it is declared. If a host is declared private, it is assumed to be distinct from identically named hosts mentioned outside this scope.

For example, suppose there are two machines named `bilbo`. If the input shows a link from `bilbo` to `princeton` and another link from `bilbo` to `wiretap`, *pathalias* builds these data structures:



If the latter instance of `bilbo` is declared private, these data structures are built instead



Of course, the two declarations must be in different files for this to succeed.

Memory allocation woes

The volume of input data presented to *pathalias* can be overwhelming. USENET maps contain over 5,700 nodes and 20,000 links, while ARPANET, CSNET, and BITNET add another 2,800 nodes and 8,000 links. Since nodes and links are dynamically allocated, the selection of a memory allocator is critical to good performance.

We experimented with several techniques⁷ covering a broad spectrum of time-space tradeoffs for memory allocation. We discovered that a buffered *sbrk* scheme for allocation, with no attempt to re-use freed space, gives superior performance in both time and space. This is due to the allocation/freeing pattern of *pathalias*. Most allocation takes place during the parsing phase, with very little space freed. After parsing, only minuscule amounts of space are allocated, while just about everything is freed. Thus memory allocators that attempt to coalesce when space is freed simply waste time (and space). For portability to segmented architectures, we use the C library *malloc* to obtain space for our memory allocator, since machines with small (64 kbyte) segments frequently impose severe constraints on the use of *sbrk*.

⁷ Implementations *f*, *b*, *s*, and *d* described in D.G. Korn and K-P Vo, "In Search of a Better Malloc," in *Proc. Summer USENIX Conference*, Portland, 1985.

Hash table management

Host names are stored in a hash table that employs open addressing and double hashing. Given a host name, we calculate an integer key k using bit-level shifts and exclusive-ors. The primary hash function is the remainder of k divided by the (prime) hash table size, T . For the secondary hash function, we do not use the oft-suggested $1+(k \bmod T-2)$,⁸ as this results in anomalous behavior (that we cannot explain); rather, we use the inverse $T-2-(k \bmod T-2)$.⁹

We cannot know *a priori* how many hosts n will be declared in the input, so we use a rehashing scheme, iteratively increasing T . When the load factor $\alpha = \frac{n}{T}$ exceeds a high-water mark α_H , a new table is allocated, the entries in the old table are inserted into the new table, and the old table is discarded. We use 0.79 for α_H , as this gives a predicted ratio of 2 probes per access when the table is full.¹⁰

The new table size can be chosen by increasing T by some factor δ , in which case the sequence of primes forms a geometric progression. If δ is too large, say $\delta=2$,¹¹ an excessive amount of space is wasted when the total number of hosts happens to be slightly more than $\alpha_H T$, for a given value of T . Since we want the new table to be "large enough," but not "too large," we experimented with an arithmetic sequence for the list of candidates for T , and employed a low-water value, α_L . When α exceeded α_H , we searched the list for a new value of T that satisfied $\alpha < \alpha_L$. This is equivalent to increasing the table size by a factor of $\delta = \frac{\alpha_H}{\alpha_L}$; for α_L we used 0.49, as this gave a value of δ close to the golden ratio. In the current implementation, we abandon α_L altogether and maintain a Fibonacci sequence of primes (more or less), which also follows the golden ratio. This simplifies the computation of table sizes without changing the behavior.

Discarding the old hash table is among the few opportunities for re-using space during the parsing phase. Rather than freeing the old tables, which can range from 4 kbytes to 32 kbytes (or more), they are placed on a list and made available to our memory allocator for later use.

CALCULATING SHORTEST PATHS

The obvious algorithm for computing shortest paths in a directed graph with non-negative edge weights is *Dijkstra's algorithm*.¹² The input is a weighted graph and a distinguished vertex, called the *source*. The output is a set of paths, one for each vertex, each starting at the source.

Dijkstra's algorithm computes minimal cost paths, where the cost of a path is the sum of the weights of the edges along the path. That is, for any vertex, the path computed by the algorithm has cost that is at least as small as any other path from the source to that vertex.

The graph described by the USENET data is sparse, *i.e.*, the number of edges e is proportional to v , not v^2 . After including data describing other networks, the graph is yet more sparse. We identify two factors responsible for this:

- Our compact representation of cliques greatly diminishes the average degree of vertices in complete graphs like the ARPANET.
- It is difficult to manage, or even to collect, a large database of connection information, such as UUCP's Systems file. Also, hosts that do have large Systems files tend to be described in the USENET data as members of a network, such as the AT&T Network

⁸ D.E. Knuth, *The Art of Computer Programming*, Vol. III, *Sorting and Searching*, Addison Wesley, Reading MA, 1973.

⁹ R. Sedgewick, *Algorithms*, Addison Wesley, Reading MA, 1983.

¹⁰ G.H. Gonnet, *Handbook of Algorithms*, Addison Wesley, Reading MA, 1984.

¹¹ As suggested in A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, Reading MA, 1974.

¹² Described in A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *Algorithms and Data Structures*, Addison Wesley, Reading MA, 1983, p. 203.

Action Central clients. Here again, the compact representation of cliques lends sparseness. We therefore use a variant of Dijkstra's algorithm suitable for sparse graphs.¹³

Simply stated, we perform a modified breadth-first search of the graph, starting at the source. However, where breadth-first search normally inserts vertices into a queue and extracts them in first-in-first-out order, we use a priority queue¹⁴ and extract vertices in increasing order of path cost. (The cost of a path is the sum of the weights of the edges along the path, subject to some heuristics described below.)

The algorithm maintains three sets of vertices: *mapped* vertices, to which optimal paths are known; *queued* vertices, for which a candidate path has been found; and *unmapped* vertices, which are not yet reachable. A queued vertex v is extracted from the priority queue if it has the smallest candidate path cost, whereupon v is marked as mapped. The neighbors of v are then added to the priority queue and the edges that brought us these neighbors are marked as participating in optimal paths. If some neighbor of v is already queued, but the path through v is shorter, we reduce the cost to this neighbor, unmark the "old" edge, mark the "new" edge, and restore the heap property.

As it happens, when the mapping algorithm terminates, the marked edges form a directed tree, rooted at the source vertex. Later, we will show how to traverse this tree and generate the paths themselves.

For the priority queue itself, we use an implicit binary heap.¹⁵ This requires a large contiguous array, but since the hash table is no longer needed and is guaranteed to be large enough, we use that space instead of allocating a new array.

Time complexity

A vertex can be inserted into or deleted from a binary heap of size k in time proportional to $\log k$. The heap can never have size greater than v , and we insert and delete vertices at most e times, where e is the number of edges, so the running time for the mapping phase is proportional to $e \log v$. Since the graph is sparse, *i.e.*, e is proportional to v , $e \log v$ is proportional to $v \log v$. Both asymptotically and pragmatically, the priority queue variant is a clear winner over the standard version of Dijkstra's algorithm, which runs in time proportional to v^2 . (Note, though, that if the graph is dense, our running time is proportional to $v^2 \log v$.)

Back links

After running of the shortest-path algorithm, it's common to find that some hosts are unreachable. Before reporting these hosts on the error output, we examine the connections out of each unreachable host, invent links from its neighbors back to the host, and continue with Dijkstra's algorithm. The resulting paths are thus generated by implication, rather than by explicit declaration.

Cost calculation

In calculating path costs, *pathalias* augments the edge weight sums with heuristics designed to avoid ambiguous routes and routes through networks that demand a gateway. Although this sullies our weighted graph model, it's consistent with our pragmatic approach to cost measures.

¹³ *Ibid*, p. 207.

¹⁴ *Ibid*, p. 135.

¹⁵ Sedgewick, *op cit*, p. 130.

Avoiding ambiguous routes

It is widely acknowledged that no simple measures suffice for disambiguating a route that contains both '@' and '!'. Although effective heuristics have been proposed,¹⁶ this approach is not widespread: most mailers rigidly adhere to "UUCP syntax" or to "RFC822 syntax."¹⁷ As such, they consistently make the wrong choice on selected inputs. It's clear, then, that we should be willing to pay a price if it results in fewer ambiguous routes, so *pathalias* adds a heavy penalty to paths that mix routing syntax.

As it happens, with our (atypically large) data set, this penalty is applied to only a fraction of a percent of the generated routes. A more significant factor in avoiding ambiguous routes is the ability of gateway hosts to accept addresses that merge domain¹⁸ and UUCP conventions. For example, where a route such as `seismo!postel@f.isi.usc.edu` was once unavoidable, it is now permissible to use `seismo!f.isi.usc.edu!postel`. We shall say more about domains a little later.

Gatewayed networks

Many hosts are situated on multiple networks and transports. For example, hundreds of ARPANET and CSNET hosts are also reachable via UUCP. However, for technical, administrative, and political reasons, only a (literal) handful provide gateway services. Therefore, we provide a way to declare networks that require explicit gateways, and a way to declare their gateways.

Any path that enters such a network through a host not declared as a gateway is severely penalized. Because hosts with domain addresses are by definition ARPANET hosts, domains and subdomains are assumed to require gateways. A similar constraint also affects links out of domains, so that once a path enters a domain, *pathalias* penalizes further links. This assures compliance with ARPANET restrictions on use of the network as a relay.

PRINTING THE ROUTES

With the shortest path tree identified, we now enter the third phase of *pathalias*. The goal is to print each host name followed by the route to that host. Routes are presented as *printf* format strings, e.g., `ulysses!decvax!%s`. A mail user or delivery agent combines this route with a user name, producing a complete route.

Routes are computed by labeling nodes in the shortest path tree in a preorder traversal. We first label the root, which corresponds to the local host, with route `%s`. In the recursion step of the traversal, we calculate the route to a child node by combining the parent's route and the routing information in the parent-to-child edge.

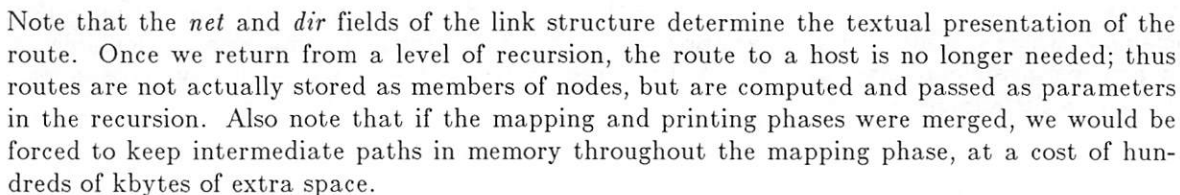
Link routing information consists of a character to use as the network routing operator, and the specification of whether a host should appear on the left or right of the operator. For example, UUCP links use '!' and LEFT, while ARPANET links use '@' and RIGHT. As the traversal proceeds, the route to a node being visited is constructed by replacing `%s` in the parent's route with `host!%s` or `%s@host`.

For example, the following tree fragment, rooted at `princeton`, has been labeled with routes.

¹⁶ P. Honeyman and P.E. Parseghian, "Parsing Ambiguous Addresses for Electronic Services," *Software — Practice and Experience*, to appear.

¹⁷ D.H. Crocker, "Standard for the Format of ARPA Internet Text Messages," RFC822, Network Information Center, SRI International, Menlo Park CA, 1982.

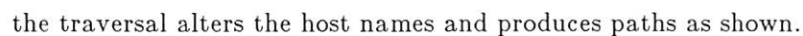
¹⁸ P. Mockapetris, "Domain Names — Concepts and Facilities," RFC882, Network Information Center, SRI International, Menlo Park CA, 1983.

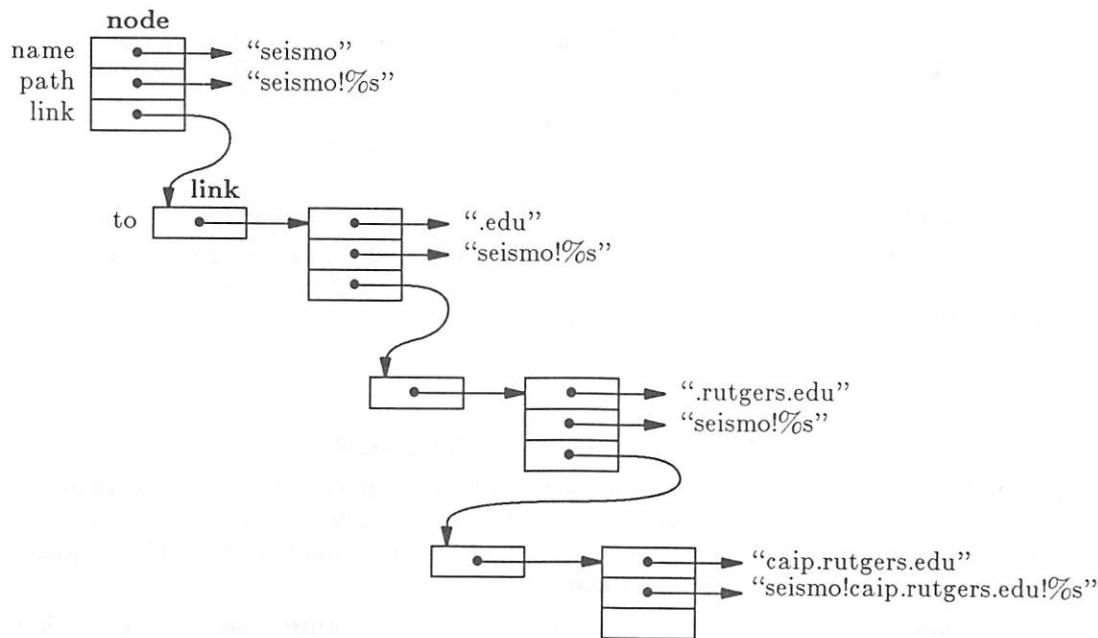


Pathalias gives networks special treatment: the route to a network is identical to the route to its parent. Except for domains, a network (and its route) never appears in the output, serving only as a placeholder for routes to network members.

Private hosts are labeled just as other hosts, but no output is printed. However, a private host name might appear in the output as a relay to another, non-private host.

Like networks, domains act as placeholders for routes to domain members. But domains play a larger role in building up routes: upon encountering a domain in the tree traversal, the name of the domain is appended to the name of its successor. For example, given the following tree fragment





As with networks, the cost from a domain to a subdomain is zero. However, the rule for building domain routes does not preclude traversals “up” the domain tree, if they happen to be “down” the shortest path tree. We therefore assume that all domains require gateways, and treat a parent domain as a gateway to its member domains. This imposes a heavy cost penalty, essentially infinite, on the edge from a subdomain to its parent, which prevents such absurdities as `caip!seismo.css.gov.edu.rutgers!%s` (which may or may not succeed — we don’t care to experiment).

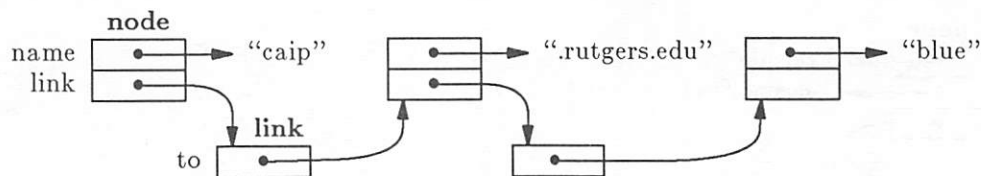
Unlike other networks, a top level domain, *i.e.*, a domain whose parent is not also a domain, is shown in the output. The route is given by the route to its parent (*i.e.*, its gateway).

As with routes to hosts, domain routes are intended to be used by a mailer. While the argument for a host’s format string is generally a user, the argument for a domain is a route relative to its gateway.

For example, in the figure above, if the route to `seismo` is `seismo!%s`, then so is the route to `.edu`. To route to `caip.rutgers.edu!pleasant`, a mailer first searches the route list for `caip.rutgers.edu`; if found, the mailer uses argument `pleasant`, producing `seismo!caip.rutgers.edu!pleasant`. Otherwise, a search for `.rutgers.edu`, followed by a search for `.edu`, produces `seismo!%s`, the route to the `.edu` gateway. The argument here is not `pleasant` (as it were), it is `caip.rutgers.edu!pleasant`, producing `seismo!caip.rutgers.edu!pleasant`, as before.

Since a subdomain has the same route as its parent, the rule for top-level domain routing subsumes subdomain routing. For brevity, then, routes to subdomains are not printed.

Note that our definition of a top-level domain, one whose parent is not a domain, allows a subdomain to masquerade as a top-level domain. This permits hosts to gateway into selected portions of a larger domain structure. For example, to augment the figure above with a top-level domain `.rutgers.edu` with gateway `caip`, we add the following fragment to the domain tree:



This makes `caip` a gateway for `.rutgers.edu`, but not for the ARPANET as a whole.

If a host has a direct path to `caip`, then the route to `caip` and `blue` become `caip!%s` and `caip!blue.rutgers.edu!%s`, respectively. Observe that `.rutgers.edu` is logically an alias of `.rutgers`, but such a declaration is superfluous.

INTEGRATING PATHALIAS WITH MAILERS

There are a number of ways to integrate *pathalias* with the rest of a mail system, each with advantages and disadvantages. Whatever course is taken, it is always advantageous to make the route database available for manual querying by users. In the simplest case, this constitutes the full extent to which *pathalias* is used in a mail system.

A second possibility is to modify user agents to make queries automatically. This allows addresses in mail headers to contain actual expanded paths, rather than eliding the information. However, many sites have more than one mailer; consistency requires that all be updated. Such changes incur a continuing maintenance burden as new releases are installed.

To avoid the extremes of full manual operation and large-scale modifications of existing software, a compromise choice is to bury the database query in a separate program that is executed by a delivery agent. This avoids complicated changes to existing code; changes that must be made tend to be simple, perhaps involving a configuration table rather than a program. If there is only one transmission channel, such as UUCP, this is certainly the method of choice. Unfortunately, this is not always the case: a host may have an SMTP¹⁹ link to some hosts but not to others. Here the router must know how to speak to each possible gateway, information that is more properly the responsibility of the delivery agent.

Which brings us to the last choice: integrating the query into the delivery agent. In some cases, this is straightforward: *upas*,²⁰ for example, permits an external program to be invoked to rewrite an address. In other cases, notably *sendmail*,²¹ complicated changes to the code and configuration table are necessary. In any event, putting the database query into the mail delivery agent means that all mailers will receive its benefits, and any network may be used to transport the mail.

Another issue that must be settled is the extent to which *pathalias* data is allowed to override a user's selection of a path. In particular, given a hideously long UUCP path (such as one generated by a USENET reply), should the mailer simply find a route to the first site in the string, or should it search for the rightmost host known to its database? The latter approach can result in significant savings; unfortunately, it can backfire if the user *wants* to use a circuitous route for some reason — say, to bypass a dead link. Indeed, it may be desirable to turn off optimization entirely. Loop tests are a time-honored UUCP tradition, and an overly-enthusiastic optimizer can eliminate them altogether.

Whatever choices are made, it is vital that any address visible in a locally-generated mail

¹⁹ J. Postel, "Simple Mail Transfer Protocol," RFC 821, Network Information Center, SRI International, Menlo Park CA, 1982.

²⁰ D. Presotto, "Upas — A Simpler Approach to Network Mail," in *Proc. Summer USENIX Conference*, Portland, 1985.

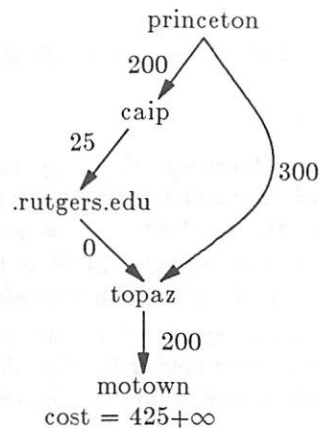
²¹ E. Allman, "SENDMAIL — An Internetwork Mail Router," in *UNIX Programmer's Manual*, 4.2 Berkeley Software Distribution, 1983.

header must be acceptable if received in remote mail; if this rule is not heeded, replies may not work properly. Strictly speaking, this rules out using first-hop routing for remote mail while using “smart” routing for locally-generated mail; in practice, this causes trouble only if the first hop on such a path isn’t known at all, but a later one is — a rare phenomenon.

PROBLEMS

We view our treatment of host name collisions with some skepticism. While we believe in individual hosts managing their own name spaces, the job of identifying private hosts is time-consuming and sometimes arbitrary. We would be pleased if, for example, the UUCP mapping project data either marked host name collisions with private declarations or simply excluded them. Even then, data collected from other sources must be perused to identify problem hosts. The only possible solution is a global absolute name space, but we see no viable candidate that can compete with the ease, low-cost, simplicity, and extensibility of UUCP’s relative address space.

A more technical problem with *pathalias* is its insistence on using paths strictly out of the shortest path tree. Once *pathalias* has decided on a route, it is committed to that route for hosts beyond it. Consider, for example, the following connection graph.



At first glance, the best route to *motown* follows the left branch, with cost 425, instead of the right branch, with cost 500. However, the domain heuristic penalizes the former route, so that the right branch should be preferred. (In practice, the mailer at Rutgers rejects the left branch route.)

The problem lies with our shortest path computation: we compute a shortest path tree, but the routes we want to generate cannot be represented in a tree. We are currently experimenting with a modified algorithm that maintains the “second-best” path when the shortest path to a host goes by way of a domain.

PERSPECTIVES ON RELATIVE ADDRESSING

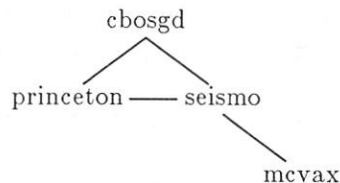
Relative addressing is employed on most networks, even some that advertise otherwise. For example, while ARPANET requires compliance to RFC822 rules, member hosts stretch the rules with underground syntax: *user%host@relay*. Although this syntax is legal, it achieves neither the ARPANET goal of pure absolute addressing, nor the UUCP virtue of consistent syntax.

Absolute addressing requires absolute compliance to standards promulgated by an absolute authority, yet it is impossible to bring absolutely every host under any authority. This is especially true when crossing administrative boundaries, *e.g.*, from the ARPANET, administered by

the Network Information Center, to UUCP, largely unadministered. Furthermore, even RFC822 recognizes the usefulness of explicit routing, and provides a (clumsy) syntax to support it. Finally, any host that hopes to provide gateway services to UUCP is forced to accommodate the UUCP addressing conventions.

With this as the framework for internetworking, *pathalias* is a tremendously useful tool. Inter-network addresses are built up by splicing together the name spaces of intermediate hosts and gateways. Without a routing tool, this places a heavy burden on users, in keystrokes and long-term memory. *Pathalias* frees users of these requirements. Yet this is a mixed blessing, as it can interfere with route translation. We illustrate by way of an example.

Consider the following fragment of the UUCP connection graph.



(All links are bidirectional.) Suppose a message is sent from `cbosgd!mark` to `princeton!honey`, with a copy to `seismo!mcvax!piet`. The message arrives on `princeton` as

```
From cbosgd!mark Sun Feb 9 13:14:58 EST 1986
To: princeton!honey
Cc: seismo!mcvax!piet
```

From the perspective of `princeton!honey`, the copy recipient is `seismo!mcvax!piet` relative to `cbosgd`, i.e., `cbosgd!seismo!mcvax!piet`, but this can be safely shortened to `seismo!mcvax!piet`.²² However, if `cbosgd` runs *pathalias*, the “carbon copy” header might be abbreviated `mcvax!piet`; the copy recipient is now `cbosgd!mcvax!piet`. This cannot be safely transformed without making assumptions about host name uniqueness.²³

Pathalias thus warps the relative name space of UUCP, providing a false sense of absolute addressing. While *pathalias* works well as a router for the physical topology of a network, it can be abused when applied to the name space topology. In the above example, message headers clearly support the view that host `mcvax` is in the name space of host `cbosgd`. But it is folly to treat this as the case, as it makes route optimization a complex problem in distributed computing, impossible to solve in a network of point-to-point, low bandwidth links.

For message headers to be useful, they must be accurate. This can be accomplished only if user or delivery agents expend some effort. While we do not advocate the approach taken by *send-mail*,²⁴ in which headers are subjected to deep analysis and complex transformations, we urge adherence to some simple principles:

- Message headers should be modified only as necessary to conform to network standards.
- Other message data should not be modified at all.
- A host must not generate a return path that would be rejected if used.
- Hosts that re-route mail from local users should show the modified routes in message headers.
- Relays within a network should not modify routes, nor translate to foreign addressing

²² See Honeyman and Parseghian, *op cit*, for details.

²³ Some UUCP hosts have adopted the ARPANET's domain syntax, and are listing their names with a central registry. Among these hosts, it is reasonable to assume names are unique, and that full routing is supported. But the situation is in flux.

²⁴ Allman, *op cit*.

styles.

- Gateways should translate between addressing styles when providing gateway services.

Compliance with these guidelines, even loose compliance, can make internetwork addressing and routing a practical reality.

ACKNOWLEDGEMENTS

We thank Rick Adams, Steve North, Pat Parseghian, and Marc Stavely for many fruitful discussions. Pat also helped a great deal with this paper. Bob Sedgewick and Bob Tarjan helped with some of the fine points in algorithm design. Paul Haahr and Dave Hitz gave sound advice to their advisor. We thank Dennis Ritchie, Dave Presotto, and Mark Horton for comments on an early draft of this paper.

For the past four years, many of our USENET correspondents have played an important role in the development of *pathalias*. Among them are Paul Bame, Marc Brader, Piet Beertema, Scott Bradner, Robert Elz, Ray Essick, Erik Fair, Stephen Gildea, Jack Hagouel, Jordan Hayes, Johan Helsingius, Hokey, Mark Horton, Sam Kendall, Gary Murakami, Greg Noel, Mel Pleasant, Guy Riddle, Larry Rogers, Eric Roskos, Bill Sebok, Chris Seiwald, Alan Silverstein, Rob Warnock, and the UUCP map coördinators. A special thanks to Karen Summers-Horton, who slaved over the early UUCP and USENET maps.

Finally, we thank the thousands of people who use *pathalias*; their reliance on our ideas has been a constant encouragement, and their palpable confusion spurred us to write this paper.

Pollster: A Document Annotation System for Distributed Environments

Luis-Felipe Cabrera

Eric Mowat

Computer Science Department
Office System Laboratory
IBM Almaden Research Center ¹
650 Harry Road
San Jose, California 95120-6099
cabrera@ibm.com.ARPA

Computer Science Division
Department of Electrical Engineering
and Computer Sciences
University of California, Berkeley
Berkeley, California 94720
mowat@ucbarpa.berkeley.edu.ARPA

Abstract

Pollster is an experimental distributed software system designed to ease the production and revision of technical documents by having them commented on by a community of people in an efficient and user friendly way. Aiming at filling the need for tools to enhance collaborative work, Pollster offers a distributed service to a (possibly very) disperse user community. Pollster's user interface relies on a pointing device to select predefined functions in the context of execution.

Care has been taken to define and implement interfaces which ease porting this software to different underlying hardware architectures.

Section 1. Introduction

In a period where office and field team work has become more pervasive there is a lack of computer tools which aid this kind of work (Furuta et. al.). Within the Berkeley UnixTM software, for example, we find no applications which support document production by groups of users. Pollster is a system which, in a series of private interactive sessions, allows a group of people to provide an author of a document with their comments and suggestions about a document in preparation. All comments made by reviewers of a document are sent to the author through the system in batch style. Thus, while supporting and easing inter-personal collaboration for document refinement Pollster provides no support for concurrent document creation by a team of users working in different sites.

¹ All of the work reported here was done while the authors were with the Computer Science Division of the Electrical Engineering and Computer Sciences Department of the University of California at Berkeley. This work was done under the sponsorship of the University of California and Xerox Corporation MICRO Grant.

A Pollster node is any node in a distributed computing environment that could be based on a local-area network or on a long haul one in which Pollster is available. We assume nodes with facilities such as editors, text formatters, and such necessary hardware capabilities as bit-map displays to obtain a faithful presentation of the formatted document. Our current software only runs on SUN/BSD environments, but we have architected the software in ways that promote porting to other environments. In any Pollster node an author can produce a document. The author may then send the document for comments to a set of reviewers, who need to be located at other Pollster nodes. Requests for comments may have attached to them review deadlines or other constraints, such as lengths of comments or type (or level) of desired comments. Reviewer's comments will be returned to the author, who will then proceed to revise his or her document accordingly.

In Pollster, a document with its annotations can be thought of as a collection of files with associated viewing windows. The files associated with the annotations of a document form a tree in the file system hierarchy. This eases the manipulation of annotations. While reading the document, the reviewer makes annotations. They remain denoted by appropriate markers in the review copy of the document. Annotations may be entered at different viewing levels: a page, a paragraph, a set of sentences, a phrase, a set of words, a character. Annotations are made and displayed through appropriate windows via menu-driven interfaces using a mouse as a pointing and selection device.

When sent to the author, the reviewer's annotations appear as markers in the review copy of the document. The author reads them by expanding each window associated with the text of an annotation. Markers from different reviewers all appear simultaneously in the author's review copy of the document. Superimposed markers can be toggled by clicking on them. Pollster provides the possibility of selective display of annotation markers. Different levels may be turned on and off at will. The author may ignore and delete annotations or insert them automatically. The modifications made on the document by the author or reviewer with editing privileges constitute a new document. Through unique document identities the system is able to collate and integrate annotations from disperse nodes in an internetwork.

In contrast to RCS and SCCS (Tichy, Rochkind, Glasser), Pollster is not concerned with preserving and tracing all successive modifications of a document. Pollster aims at aiding an author in creating and reviewing a document, in circulating a review copy for comments, in gathering all of these comments, and in allowing the author to incorporate comments made by third parties. Pollster should be viewed as a text processing manager which owns and administers files given to it by authors.

Pollster provides tailored environments for three kinds of events: the creation or writing of a document, the revision or commenting of a document, and the utilization by an author of the reviewer's comments about a document. Traditional text processing facilities (Carmody et. al., Chamberlin et. al., Furuta et. al., Knuth, Kernighan, Meyrowitz and van Dam) have addressed these issues. Our architecture uses existing underlying text processing software and superimposes a mechanism for displaying, annotating, and transporting a document and its associated comments.

The rest of the paper is divided as follows: in Section 2 we describe Pollster's architecture. Section 3 displays figures which illustrate the user interface, while Section 4 relates our initial experience with the system. Section 5 describes future work and Section 6 has our conclusions.

Section 2. System Architecture

Pollster has three major subsystems: document creation, document reviewing, and document shipping. We describe them in turn.

Section 2.1. Document Creation

The system has two basic modes of operation for document creation: an author submits to Pollster a document which has already been created, or an author uses the facilities within Pollster to

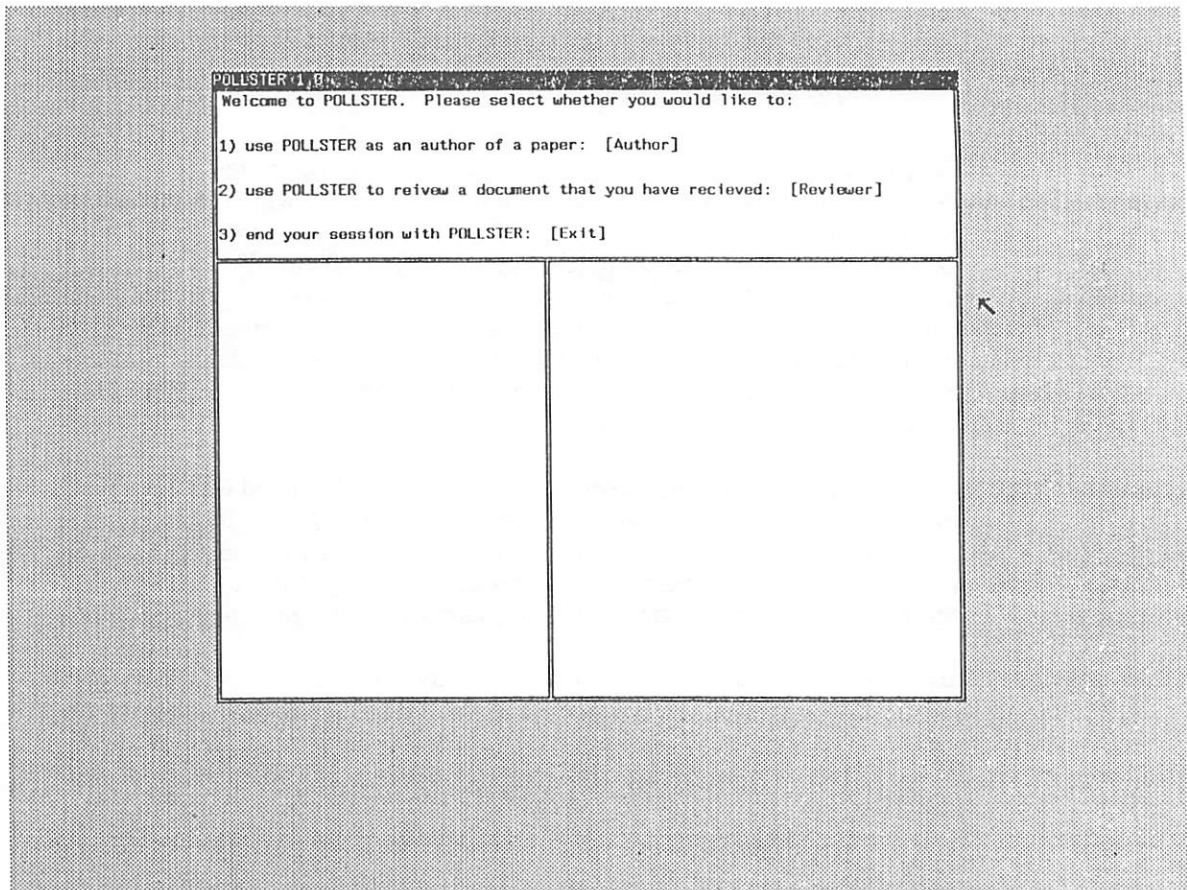


Figure 1: Main Pollster Menu. There are no other SUN windows open at this time in the system. From this window the user selects whether to use Pollster as a reviewer or as an author. The user also has the option of exiting from Pollster.

create a document and eventually submits it. In our current implementation Pollster simply offers a Unix shell to allow the user to create his document. When the user finally submits a document to Pollster, the source file for it is assigned a unique identifier. The user is additionally prompted the device independent version of the document suitable for display, for example the *ditroff* form of the document. It is this device independent display version of the document which is shipped between Pollster nodes.

The author then requests the system to send out the document for review by providing a list of user names with their host identities. The system will accept UUCP and Darpa's Internet styles of addresses. The author further specifies which reviewer has editing authorization to create new versions of the document. Pollster will also send the original source file to those reviewers with editing authorization. An author can further request that the reviews be back by a certain deadline. Pollster forwards this information to the node in which the user will review the document and retrieves the document from the reviewer's set of documents by the expiration deadline, if such constraint exists. Those reviewers who do not have editing authorization will not be able to create new documents derived from the original document, nor keep a copy of it.

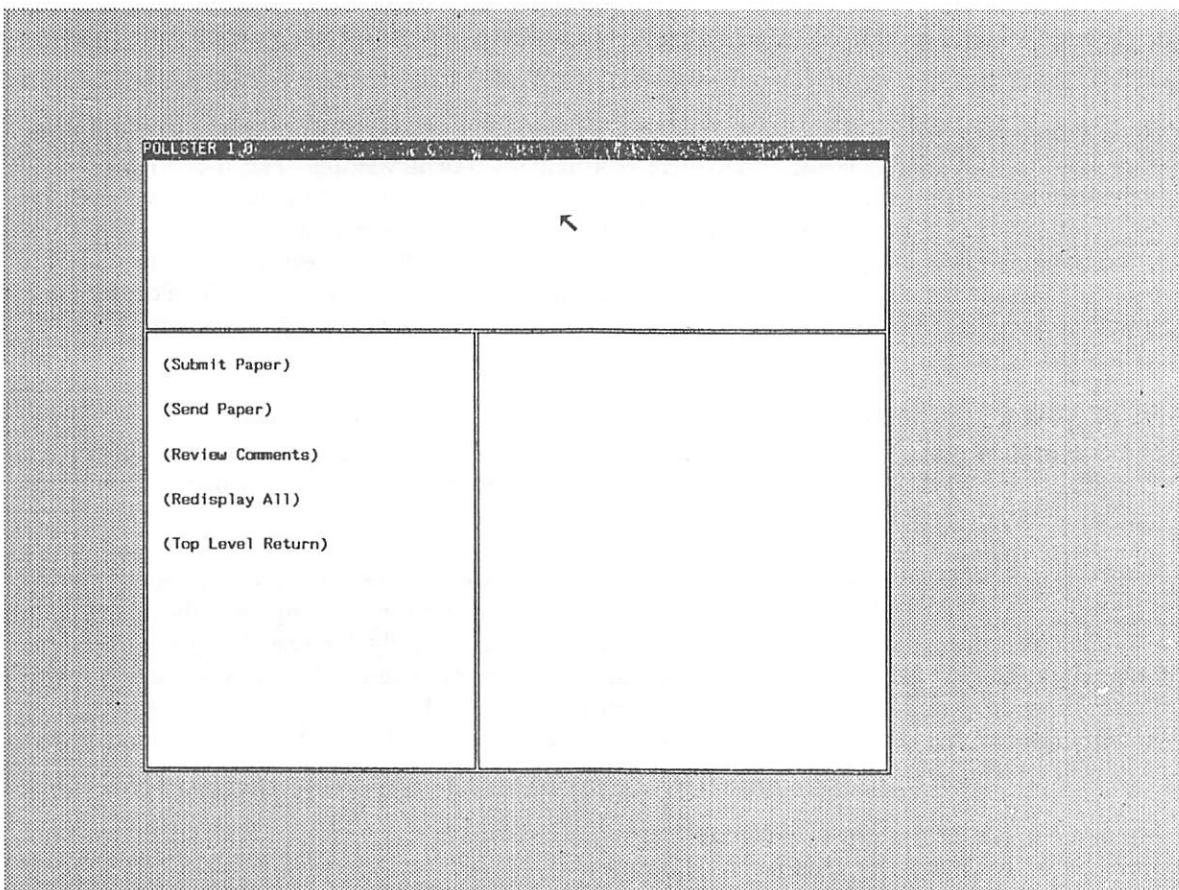


Figure 2: Top menu for an author. There are 5 possible actions: submit a paper to Pollster, send it, review a paper, redisplay the screen, and return to the main Pollster menu.

At any point in time after a document has been sent out for review an author may request Pollster to send it to another reviewer. There are no limits to the number of reviewers who may be asked to work on a document. When a deadline has been stated, authors can have a very good estimate of the number of reviewers which worked on the document after a transmission delay threshold has gone by. All document transmissions are positively acknowledged. Reviewers may not forward a document to third parties.

Inter Pollster node authentication, message routing, and reliable message delivery are all based on standard 4.3BSD utilities. We shall come back to them when discussing document shipping.

Section 2.2. Document Reviewing

The two main software components of document reviewing are the document displayer and the annotation handler. Our current prototype only offers displaying capabilities for documents which have been formatted using *ditroff*. The heart of displaying is the program *dsun.tool*² which is a version of *dsun* able to display documents from within the window subsystem. An analogous displaying tool for *Tex* based documents exists, but we have not incorporated it into the prototype. The annotations handling software, as described below, is independent of the display software.

When a document arrives at a node for review by a user who has an account in that node, Pollster sends mail to that user indicating that there is a new document waiting. The user then invokes Pollster and requests to see the list of documents which require attention. The reviewer has the option of saying "no time available" for a given document. Pollster then sends a message back to the author's node and proceeds to end the reviewing process for that document. The end of a reviewing process for a document involves destroying all relevant files created in this Pollster node as well as updating the internal state of the node to reflect that there is no pending interest in this particular document.

When a reviewer is finished with a document, he sends an "end of review" command to Pollster. The system then proceeds to transfer to the author's node all the relevant information. Only when an explicit acknowledgement of the shipment is received does the reviewer's node update its internal state to show the review is complete.

A normal reviewing process is done in (perhaps many) sessions. The review session is done using the review window which will display pages of the current version (see Figure 4 through Figure 11). Complete formatted pages could not be displayed exclusively for readability reasons. The fonts would have to be too small. The current display monitor resolution (in combination with the fonts we had available) is not good enough to accommodate this. A reviewing session may be terminated at any time and resumed later. All (earlier) comments made by a reviewer, on any page, will be available for inspection and/or modification throughout the reviewing process.

Annotations can be entered at different viewing levels: a page, a paragraph, a set of sentences, a phrase, a set of words, and a character. There is also the transparency level, which allows the user

² This program is independently available for general use.

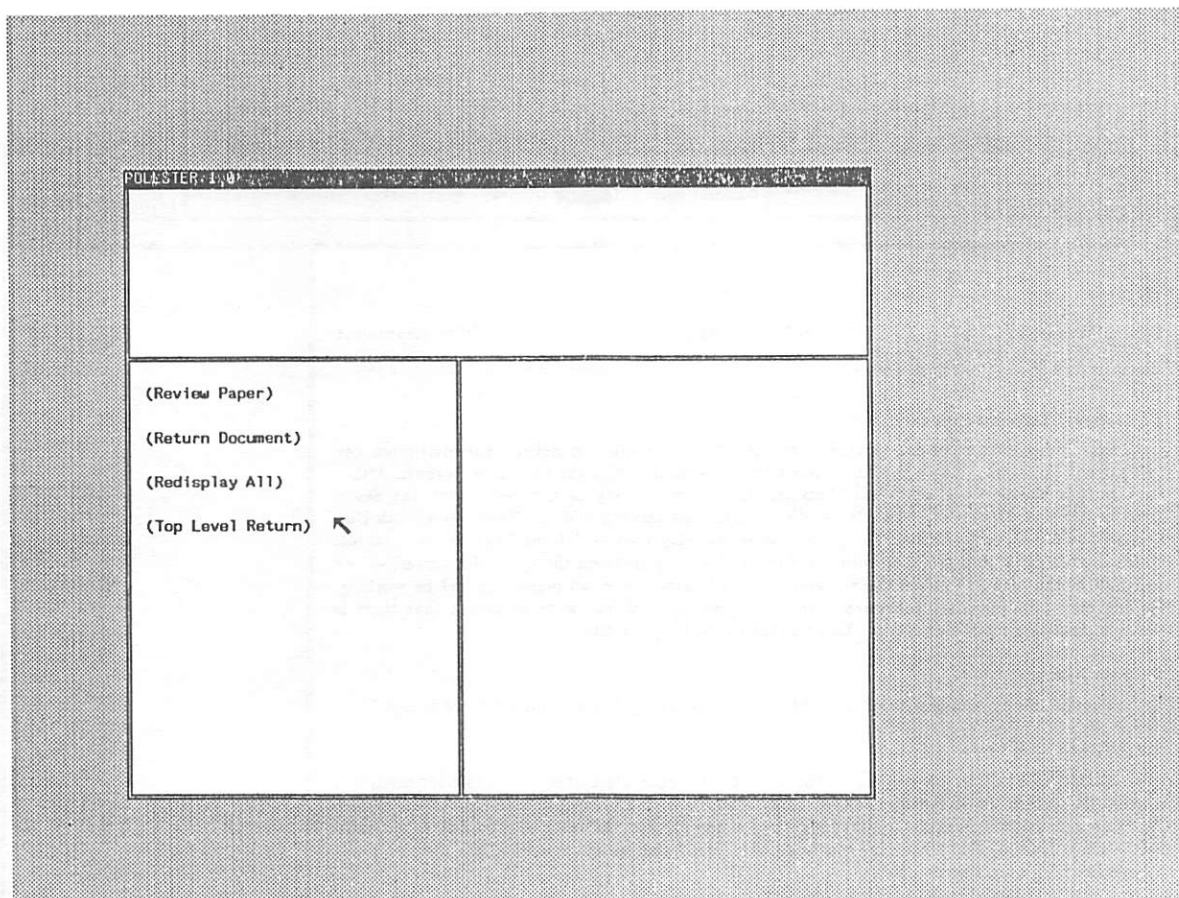


Figure 3: Top menu for a reviewer. One may either review it, return it, redisplay the screen, or return to the top level main Pollster menu.

to draw annotations over the text, as with a pencil. This mode can be used to make the typical editing marks indicating exchange of letters, deletion of words, etc.

In all levels but the transparency one, annotations are denoted by appropriate individual markers and delimiters. The transparency level appears superimposed to the formatted document. Annotation markers and annotation region delimiters both appear in the page being displayed at the review viewing window. The markers are icons designed to show the level of the comment, and to present a physical place where a user may point to and click with the mouse. All of these marks scroll together with the underlying text. Their placement is relative to the displayed area of a page of a document.

A reviewer generates an annotation using a mouse driven menu by specifying the physical limits in the text of the pertinent document area and the level of its scope (see Figures 6, 7, and 8). Once these two actions are done, a standard *vi* window is presented so that the comments are entered. An "end of annotation" command causes Pollster to make the commenting window disappear, affix permanently the marker and delimiter in the viewing window, and generate the appropriate field in a file to store the comment. The markers are icons designed to show the level and identity of the annotation, and to present a physical place where a user may point to with

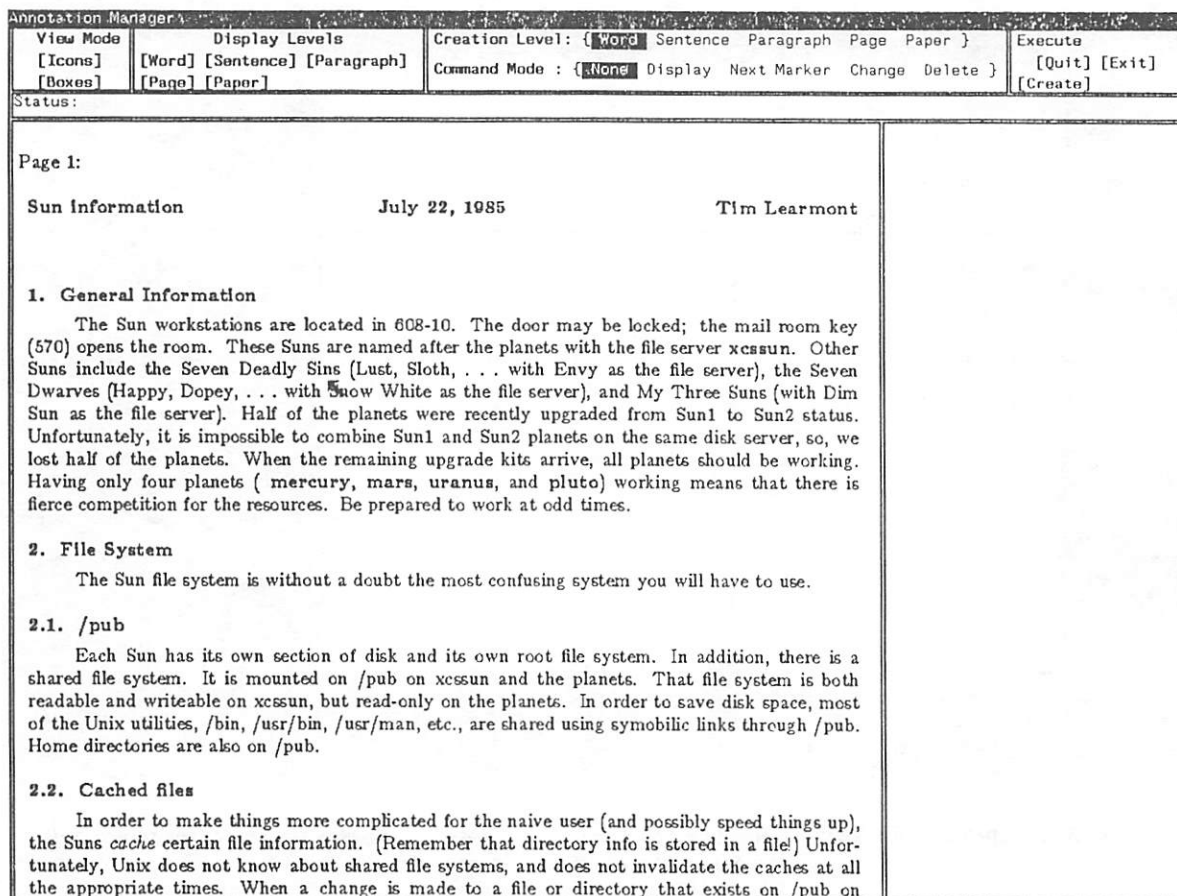


Figure 4: A reviewer is browsing at the top of the first page of a document. No View Mode selections have been made.

the mouse in order to request the display of the pertinent annotation. Reviewers may, if they so wish, delete a previously made annotation. All visible traces of it, the delimited region in the document as well as the marker, will disappear.

If a reviewer has editing authorization, then the document may be modified. In this case a copy of the source file will have also been sent. All modifications generate new documents, whose source file is sent back to the author at the end of the annotation process. The author reconciles these two documents using the standard editing facilities.

Section 2.2.1. Using the Reviewer's Comments

Once the annotations have returned, an author is notified via mail at the Pollster node where the document was submitted. We are implementing an option to notify authors by mail at a node of their liking. The author may then make use, at the Pollster node where the document was submitted for comments, of all the reviews. Pollster places in the appropriate tree the comments as they arrive from the reviewers. The underlying tree of files, though, is never to be touched by an author.

To the author, all the reviewer's annotations appear as markers with an associated text delimiter perimeter. The author reads them by expanding each window associated with the text of an annotation. Markers from different reviewers all appear simultaneously in the author's review copy of the document. Authors have the alternative of selecting which levels of annotations to see and of which specific authors. In any given page of the document the author may do this selection. Using standard editing facilities, or the *stuff* command of SUN's window subsystem, the author may insert in the source file, for the new revised document, those parts of comments of use. The author may discard annotations at will.

When in this mode authors use the review viewing window, the annotation windows, and also a vi window to create the revised version of the document. These should be initialized properly by Pollster and the session will have to overlay them appropriately. In short, all the author does is to modify a (copy of a) source file of its own to create a new document.

Section 2.2.2. Underlying File Management in Pollster

Each Pollster node administers a complete hierarchy of files for its own operation. Each user, author or reviewer, which has an active document, i.e., either in preparation or for review or submitted for comments, has a directory in his name (or with a unique network wide user identifier). Let's call them $name_1$, $name_2$, ..., $name_i$, ..., $name_K$. Within each $name_i$ directory, there will be directory entries for each document which requires that user's attention, and a description file specifying which of the documents are for revisions, which are being created, and which have been submitted for comments. Let's call these entries doc_1 , doc_2 , ..., doc_j , ..., doc_L , and $docDescription$.

In each of the doc_j directories one finds one file per (review version) page of the document. Let's call these pag_1 , pag_2 , ..., pag_k , ..., pag_M . These are created as soon as the reviewer makes an annotation about the corresponding page of the document. Moreover, within each doc_j directory, when appropriate, one also finds the source of the document a format independent reviewing versions, e.g., the *ditroff* version of the file in Unix, a file containing the constraints, and other editing and formatting related files. There may be other files, say in the case of an author, whose purpose is to be combined to form the source file. For example *refer* bibliography files, or *gremlin* ones. One may also have *spell* related files.

Within each of the pag_k files, there will be fields which will contain all the annotations done by a reviewer. Each annotation will have a system wide unique identifier so that it may be appropriately

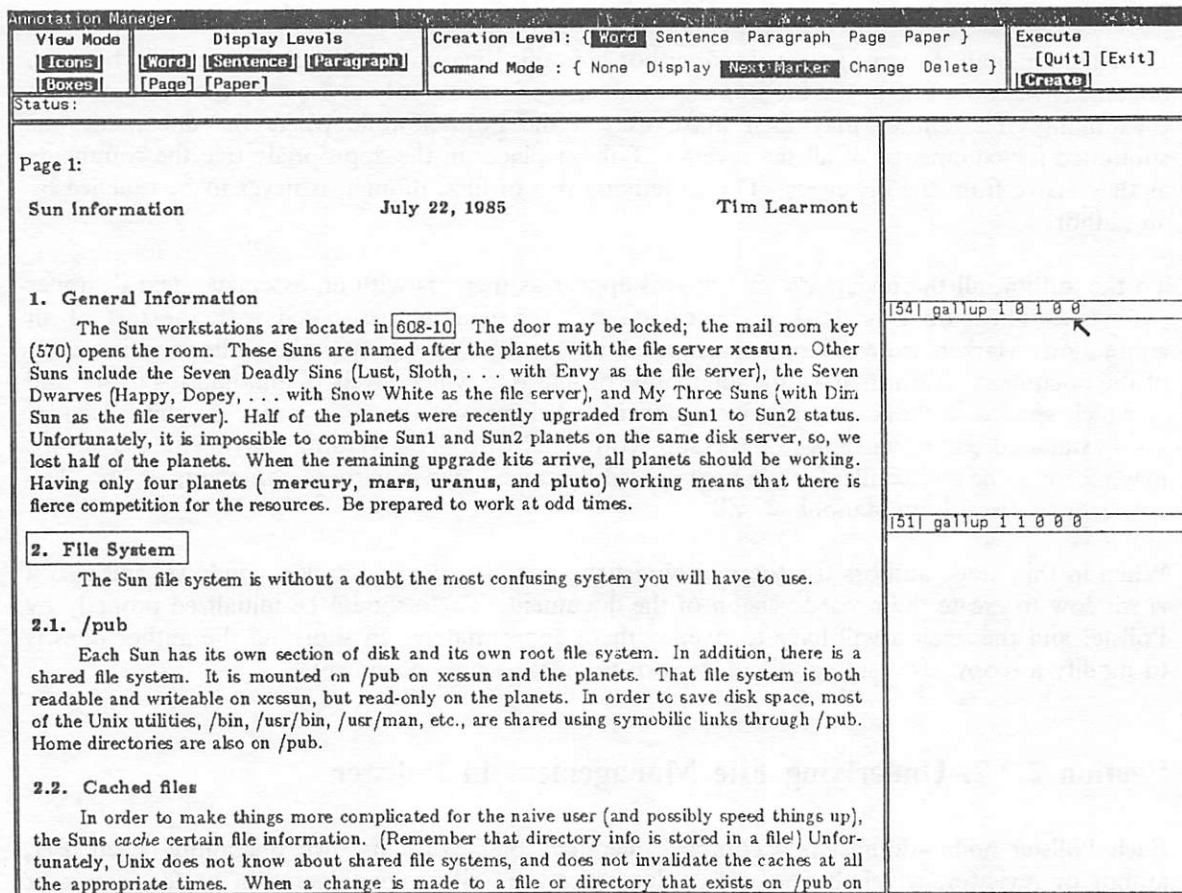


Figure 5: A reviewer is now browsing at the top of the first page of a document with the word, sentence, and paragraph Display Level selections turned on. Icons and boxes View Modes have been selected.

retrieved and displayed. The identification of annotations is displayed in the margin markers to allow facilitate referencing to them from within other annotations. The annotation description fields include the position and shape of the area delimiter, as well as the annotation level and text.

Pollster will send the appropriate page-level files back to the author when the reviewer signals "end of review". Unless the reviewer has editing privileges, these are the only files sent back. They will be put in the corresponding place in the author's node. For internode transfer, a naming scheme such as: name.doc_id.page would do. If necessary, Pollster places in a temporary directory files which have been received from other nodes but not yet placed in the tree hierarchy.

Section 2.3. Document Transmission

Upon receipt of a document for comments, Pollster first checks that the reviewers are known in the different nodes. In our BSD implementation this is done using the password file of the receiving Pollster node. When positive identification has been established for a recipient, the display file (and possible the source file) is forwarded using a robust (albeit possibly slow) file transfer protocol (*ftp* or *mail*). The sending node assumes a file has arrived at its destination only after receiving explicit acknowledgement for that file from the receiving end of the transmission. There are built-in retry and timeout constants to handle transmission errors. Pollster keeps state in stable storage regarding pending messages.

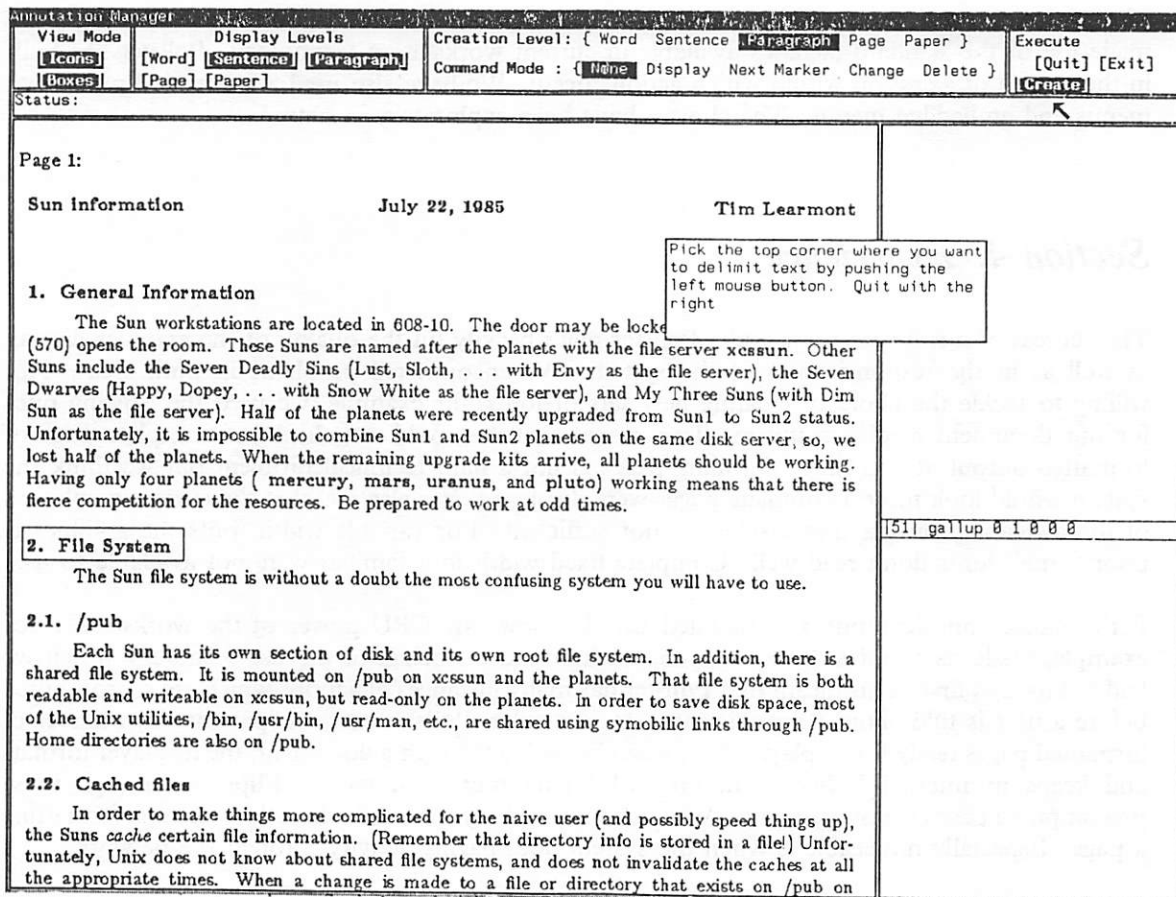


Figure 6: The reviewer has turned off the word Display Level, and is creating a new annotation at the paragraph level. Notice that the icon and the box of the existing annotation at the word level have disappeared.

If deadlines for reviewing exist, Pollster monitors the appropriate dates and sends mail messages to the author and reviewer(s) accordingly. These messages include "reminder to review", "review time elapsed", "reviewer has no time available", and "reviewer did not review on time". Date deadline translations across time zones is achieved using GMT.

The internode communication subsystem has been designed so that document annotations can be sent transparently across different underlying hardware architectures. Annotations have a standard ascii format representation which may be interpreted at any Pollster node independent of the transport medium and architectures of the sending and receiving Pollster nodes. The mail messages sent between Pollster nodes contain information that allows the decoding of the tree of annotations included in the message. We have adopted an ascii based message format to promote exchanges between heterogeneous environments.

Section 3. User Interface

The 11 interspersed figures display the basic user interface Pollster presents the user. We have made use of the standard facilities available in current workstation technology. Pollster has built in the notion of a session within which actions occur. We have also used a mixture of pull-down menus and embedded menus. The choices have been exploratory in nature.

Section 4. Experience

The success of a software project like Pollster relies heavily on the quality of the graphics display as well as in the responsiveness of the system. We encountered problems in both areas. Not willing to tackle the chore of creating new sets of fonts, for example, we used the existing ones for our document display software. This meant that we could not display a complete page of formatted output at one time. Scrolling pages is not a hard technical problem but we think the system would look nicer if complete pages were displayed. We also feel that the current resolution of the available non-graphics displays is not sufficient. For variable width fonts the graphics is poor. Small fonts don't read well. Complete fixed width font families were not available to us.

Performance considerations also haunted us. The low raw CPU power of the workstation, for example, made us adopt a "process in the background on behalf of the user" strategy which we had not used at first. This meant that Pollster has many instances of actions which are precomputed before a user is told about their existence. In particular, Pollster keeps a (parametric) number of formatted pages ready for display. Thus, when browsing through a document, the displayer formats and keeps in internal buffers a number of higher numbered pages. Flipping through these precomputed pages is instantaneous. We wish we could say the same when the system is formatting a page. Especially noticeable is when the system does paging activity through the network.

The high cost of process creation in BSD also affected some of our initial designs. In particular, the annotation display subsystem was originally implemented by *forking* on demand and *execing* an editor process which was passed the appropriate file for display. This proved to have unac-

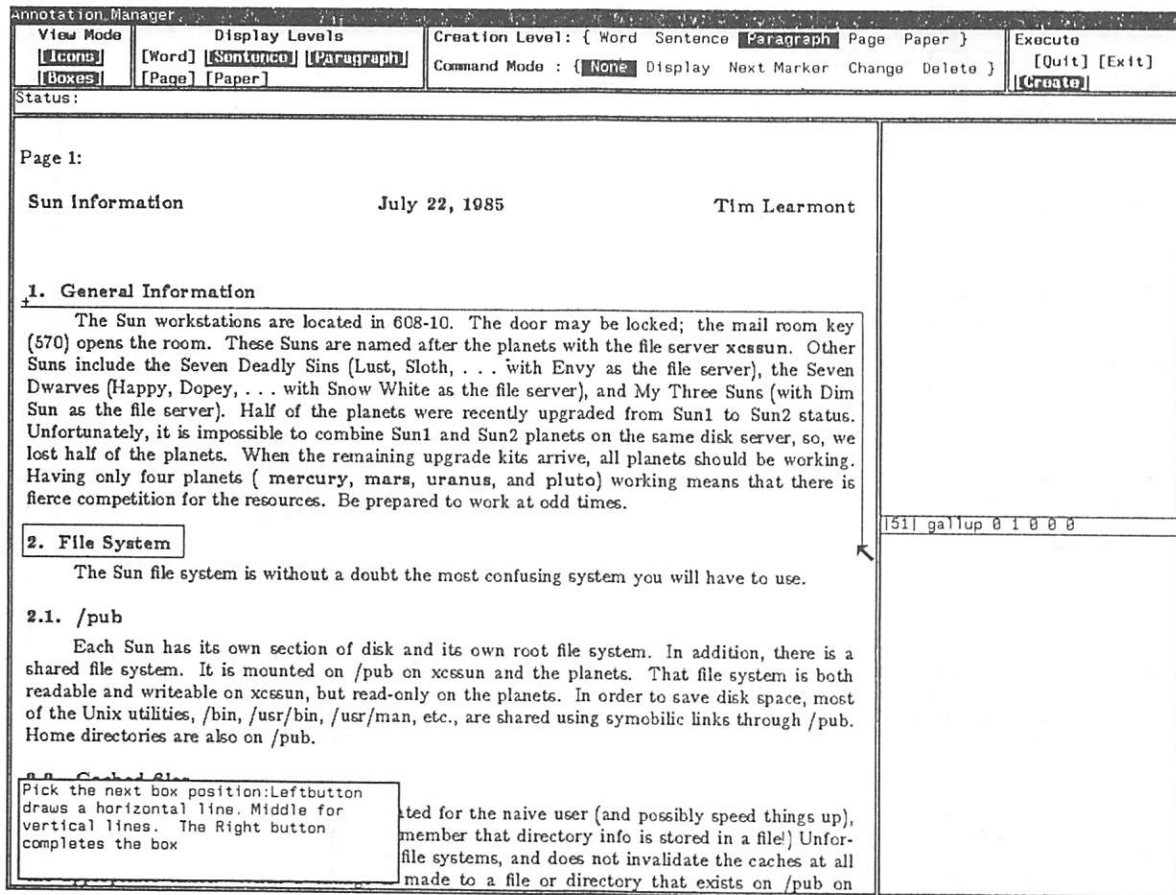


Figure 7: Pollster prompts the user successively until the delimiter box of the annotation to be created is drawn. This new annotation will be at the paragraph level.

ceptable responsiveness. The current implementation keeps an editor display process instantiated at all times and passes to it the necessary data.

Software bugs (and features) in the window subsystem also took a toll in the development time. In particular, dealing with option subwindows proved to be quite nontrivial. There is an error condition when a procedure deletes an option item, for example, as it gets repainted after returning. There is another error when checking for removed option items. When there is only one left the check is not appropriate. Some of these bugs have been fixed in recent releases of the system.

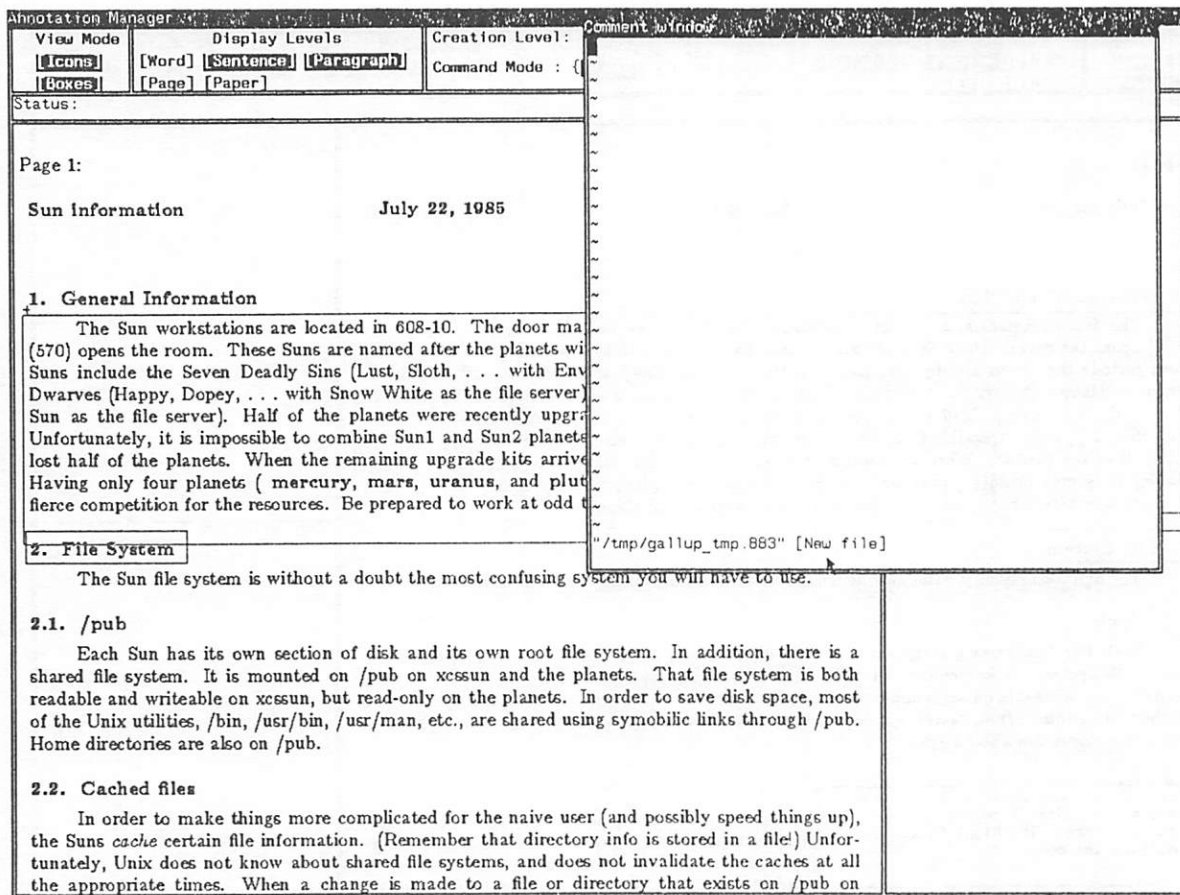


Figure 8: Once the box is complete a vi window appears so that the user may write the annotation.

Section 5. Future Work

Our current effort is centered on developing secure message exchange mechanisms, implementing a better user interface, and gaining experience with the system's behavior. A comprehensive set of icon-based mouse-operated functions is being refined and expanded to provide the user with all the standard functions of document annotation. Today only a small group of workstations has the software available. We expect to provide a larger community of workstation clusters with this software in the near future.

Our current prototype does not have facilities to allow a document to circulate with its annotations among a set of reviewers. Only the author can see the collective set of annotations done by a group of reviewers. Pollster should include an option which would allow a document to be send

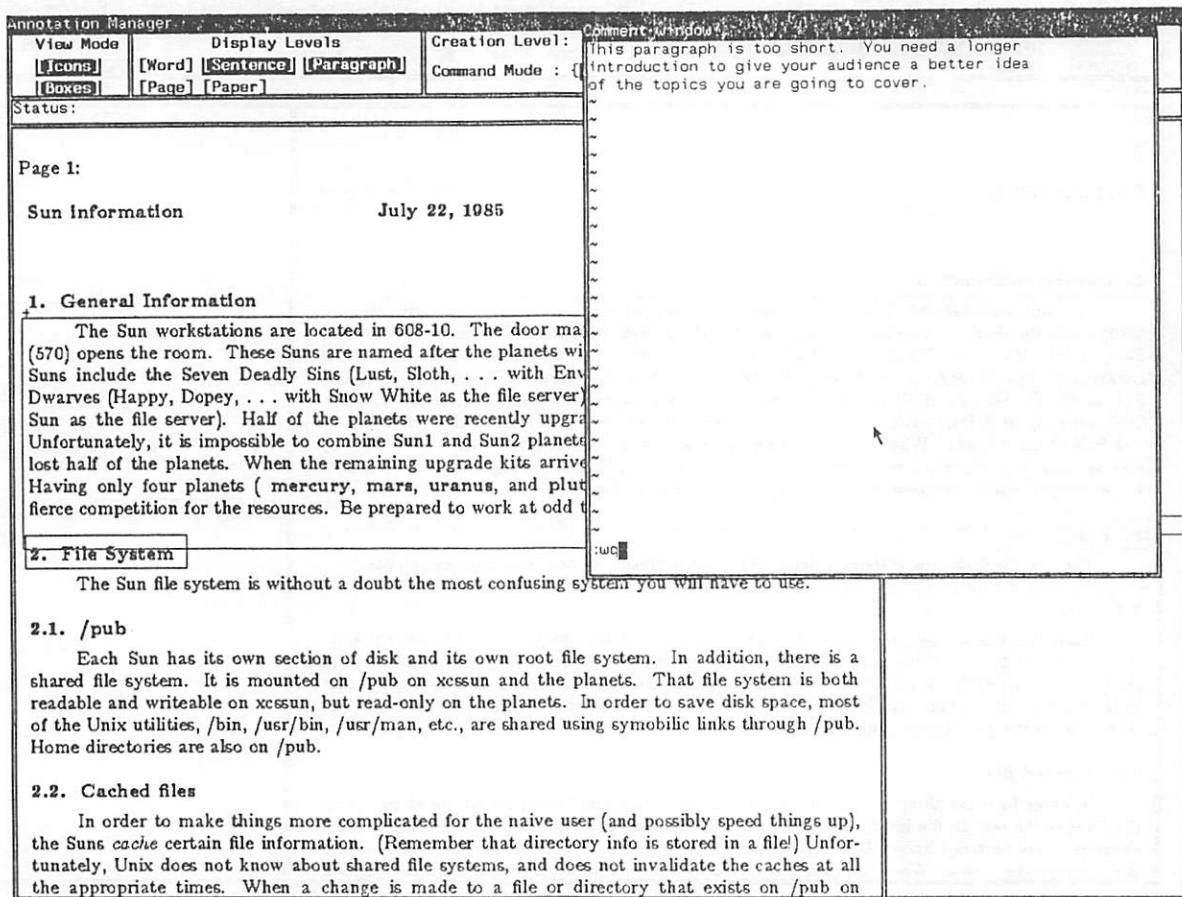


Figure 9: The user enters the desired annotation text and eventually does a *wq*.

with a set of annotations to a reviewer. This facility would also have to deal with the problem of cross references among user's annotations. Annotation identifiers displayed by the system may have to include the reviewer's identity to disambiguate them. The presentation of all of this information requires experimentation.

Another possible future step is to port the system to different architectures. This will require developing document viewer software, as well as interfacing with existing window manager sub-systems.

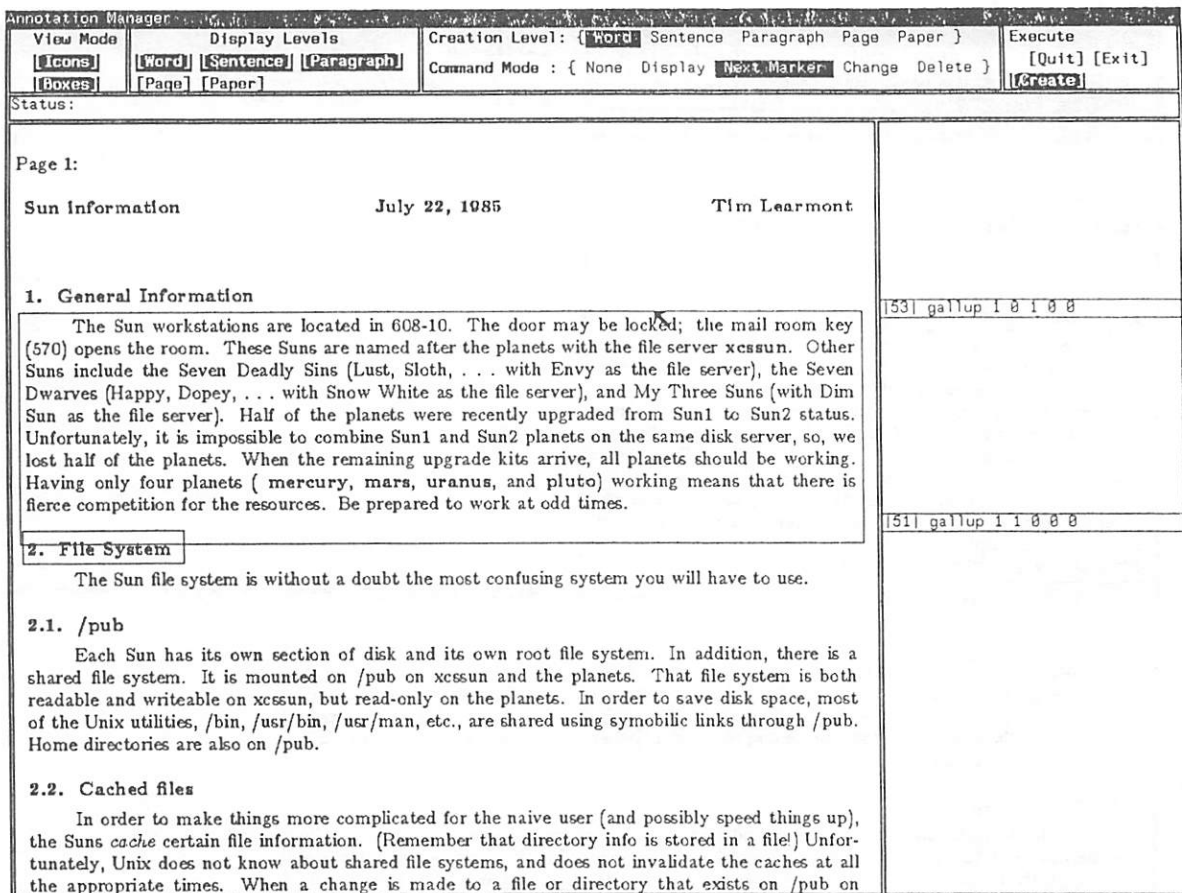


Figure 10: The newly created annotation now has a unique identifier (53 in this case) and a marker that denotes it.

Section 6. Conclusions

Pollster is a software system which provides a distributed service to a community of users which collaborate in the preparation and revision of a document. An author sends a document for review to a set of reviewers. Each reviewer, who needs to work on the document at another Pollster node, annotates the document in as many review sessions as needed. Annotated documents are then sent back to the author who can view all annotations in an integrated way.

While providing all the above described functionalities could have been achieved with underlying hardware technology of the seventies, the advent of economic workstations and networking infrastructure only now makes this software possible to use by large communities of users. In fact current graphics quality was not available at such low cost before. Moreover, the proliferation of

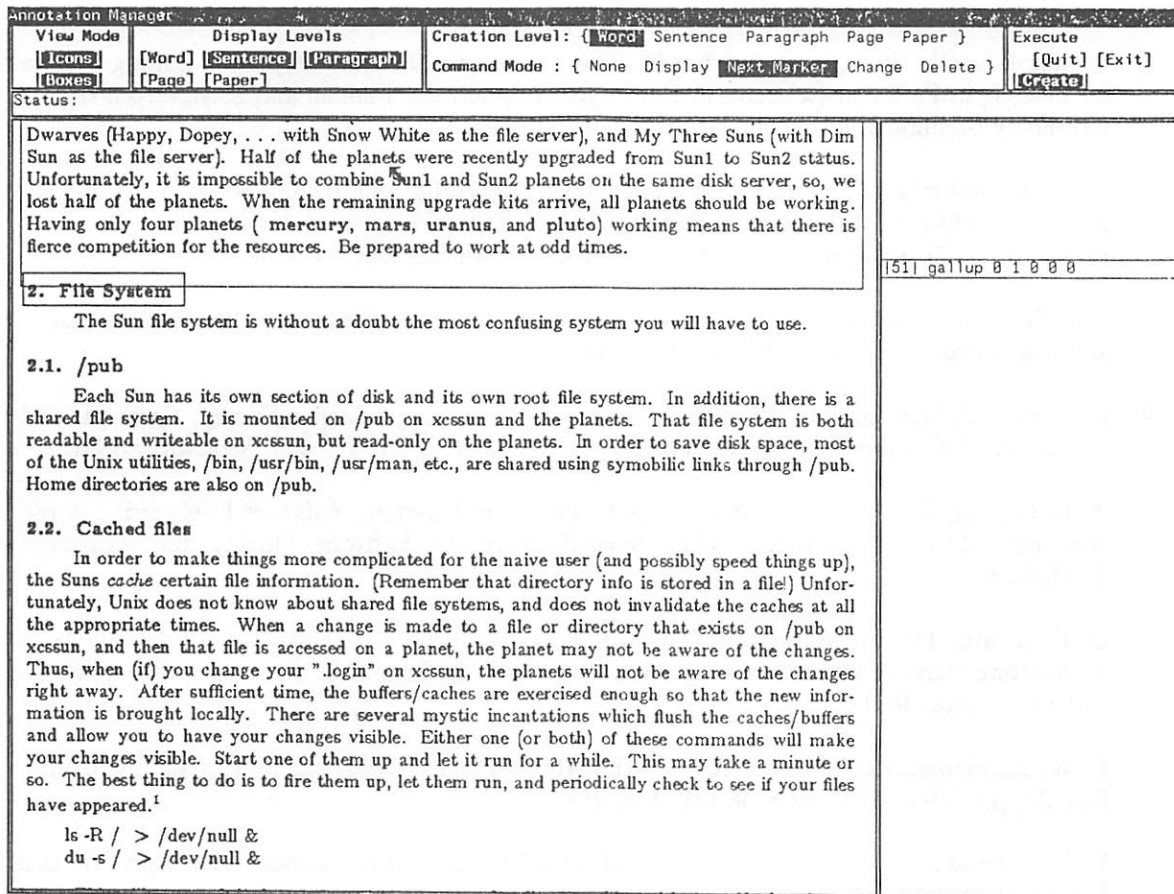


Figure 11: When the user scrolls the text, all icons and boxes scroll as well.

high resolution electronic printing devices and of bit-map displays allows an author to view documents in the workstation's screen in a format quite similar than the one it will have in printed form. This latter capability was not available a decade ago.

Pollster, albeit being a tool which operates in a stylized batch mode, promotes collaborative work in a distributed environment. There are few utilities which promote this in spite of the general trend of increased team work.

Acknowledgements. This project was developed under the auspices of a research grant of Domenico Ferrari. His suggestions helped refine many of the project's original ideas. Brian Marsh wrote the document display subsystem. Brian Bershad was an important part of the original implementation team. Our prototype would not be what it is today without their contributions. Last but not least thanks to Mike Goodfellow who provided helpful comments and careful reading of earlier versions of this manuscript.

Section 7. References

1. S. Carmody, W. Gross, T. E. Nelson, D. Rice, and A. vanDAM, A Hypertext Editing System for the 360, in *Pertinent Concepts in Computer Graphics*, M. Faiman and J. Nievergelt (Eds.). University of Illinois, Urbana,, Ill, pp. 291-330, 1969.
2. D. C. Chamberlin, J. C. King, D. R. Slutz, S. J. Todd, and B. W. Wade, JANUS: An Interactive Document Formatter Based on Declarative Tags, IBM Comp. Sci. Res. Rep. RJ3366 (40402), IBM Research Lab., San Jose, California, January 1982.
3. S. I. Feldman, Make - A Program for Maintaining Computer Programs, *Software - Practice and Experience*, 9(3), pp. 255-265, March 1979.
4. R. Furuta, J. Scofield, and A. Shaw, Document Formatting Systems: Survey, Concepts and Issues, *ACM Computing Surveys*, Volume 14, Number 3, pp. 417-472, September 1982.
5. A. L. Glasser, The Evolution of a Source Code Control System, *Software Engineering notes*, 3(5), pp. 122-125, November 1978. *Proceedings of the Software Quality and Assurance Workshop*.
6. D. E. Knuth, TEX, a System for Technical Text, in *TEX and Metafont: New Directions in Typesetting Part 2*. Digital Press and the American Mathematical Society, Bedford, Mass., and Providence, R. I., 1979.
7. B. W. Kernighan, Review of TEX and METAFONT: New Directions in Typesetting, *Comp. Rev* 22, pp. 299-301. Review 38,151, July 1981.
8. B. W. Kernigan, A Typesetter-independent TROFF, *Computer Science Tech. Rep. 97*, Bell Laboratories,, Murray Hill, N. J., 1981.
9. N. Meyrowitz, and A. van Dam, Interactive Editing Systems: Part I, *ACM Computing Surveys*, Volume 14, Number 3, pp. 321-352, September 1982.
10. N. Meyrowitz, and A. van Dam, Interactive Editing Systems: Part II, *ACM Computing Surveys*, Volume 14, Number 3, pp. 353-415,. September 1982.
11. M. J. Rochkind, The Source Code Control System, *IEEE Transactions on Software Engineering*, SE-1, pp. 364-370, December 1975.
12. W. F., Tichy, Design, Implementation, and Evaluation of a Revision Control System, *Proceedings of the 6th International Conference on Software Engineering*, IPS, ACM, IEEE, NBS, pp.58-67, September 1982.

When Network File Systems Aren't Enough: Automatic Software Distribution Revisited

Daniel Nachbar

Ⓐ Bell Communications Research
Morristown, New Jersey

ABSTRACT

The system described, named **track**, addresses the problem of maintaining software and data on multiple machines running the UNIX operating system. Under **track**, updates are made from a central *librarian* machine (or machines). All updates are initiated by the receiving machine. Local initiation of updates allows system administrators to take advantage of centrally maintained software without relinquishing control of when and how updates are made. **Track** can also map file names when making copies and execute an arbitrary shell script after a copy is made. Given these features, **track** can automate common system administration tasks (e.g. installing new releases of software) as well as difficult tasks (e.g. booting a new UNIX kernel).

System Administration in a Changing World

Only a few years ago, most computer research facilities consisted of one or a few CPU's. In the recent past, this has changed drastically. For instance, in our laboratory at BELLCORE, the computer center now contains approximately 40 CPU's of 4 different architectures running 3 different flavors of UNIX. This growth in the number and variety of machines threatens to increase greatly the amount of human effort needed to handle even the simplest tasks of system administration.

Adding to system administration problems is the fact that UNIX itself has grown. A "complete" 4.2BSD system consists of about 6,000 administration files. (This includes all the binary, library, and data files but does not include source code files.) Regularly used software packages, contribute an additional 4,000 files.

However, the large number of files that make up UNIX does not in itself create a problem. Clearly, if the contents of these files never changed, one could simply initialize them when the system is set up and forget about them. Unfortunately, at least in a research environment, these files change constantly. Some files change frequently (/etc/passwd changes almost hourly on our systems.) It is relatively easy to deal with these few, frequently changing files; of greater difficulty is the much larger mass of less frequently changing files. In our computer center, roughly 200 of these 10,000 files are modified each month. Other complexities arise because some files are shared by machines of differing architecture, while others are not. Tracking a moving target consisting of some 10,000 sporadically changing files on several machines is far too complicated a task to be done by hand.

If some sort of network files system or other information server is available, the situation is greatly eased. However, practical considerations often preclude the use of such systems for administration files. First of all, communications overhead becomes prohibitive for all but the smallest CPU's operating on a relatively fast communications channel. Second, political considerations make it highly unlikely that the administrator of one computer center would be willing to rely totally upon some other computer center for all his administration files. Today one finds

single-user workstations that are part of the same computing facility sharing administration files via an ethernet. However, it will be a long time before we see a supercomputer that is owned by a private company in New Jersey access its `"/bin"` directly from disks in Berkeley.

Thus, given that there will probably always be multiple instances of administration files, the problem is to make sure that all such instances contain the same data. To make the problem more manageable, one instance is usually designated as the *master* version of which all other machines have *copies*. While this restriction may not be acceptable for distributed data in general, it is reasonable for most system administration problems. The question then becomes one of deciding when to update the *copies* given that the *master* version of a file has been changed.

Previous Systems

Programs that address the problem of maintaining multiple instances of files on different machines have generally been referred to as "automatic software distribution systems". Several such systems have been written (we refer in particular to the `asd` system by Koenig[1] and the `rdist` system distributed with 4.3BSD.) However, one finds that these existing systems are usually operated only within a computer center. In particular, they require a central administrator (or process) to specify that a given file should be broadcast or "shipped out" to other machines. Central administration has several drawbacks. First of all there are some practical problems. There must be a tight coupling between the changing of the *master* version and its broadcast to other machines. If a *master* version is changed but accidentally not broadcast or if the update is lost in transmission, the *copies* will diverge from the *master*. Such divergence is difficult to detect. Furthermore, once divergence has appeared, it will remain until the *master* version is broadcast again.

The problem of possible accidental divergence caused by centralized administration can be overcome by periodically inspecting the *master* and *copies* of each file and making updates if differences are found (the `rdist` system uses this approach.) However, periodic checking for divergence does not overcome another, potentially more serious problem with centralized administration of updates: the loss of control by local administrators as to how and when updates are made on their machines. For whatever reasons, local administrators (in particular people who administer their own workstations) usually wish to intervene in the updating process. For example, if a machine is being used for data collection over an extended period of time, the administrator may wish to defer updates until after the study is complete. Such intervention varies from person to person, file to file, and even from time to time. Local administrators are often reluctant to allow other people to change administration files on their machines.

There is a huge range in the amount of control that local administrators wish to have. In some cases, no intervention is needed. Updates can be fully automated. This is usually the case when one person or group is administering both the source and destination machines. In other cases, a local administrator may wish to have an update take place automatically and be notified of the update after the fact. This allows the administrator to localize problems that may occur from the installation of new versions. Other administrators may wish to have new versions of files arrive in some innocuous place where they can be inspected before installation. In still other cases, an administrator may wish to be merely notified that a new version of some file is available. Central administration does not lend itself to such local customizations. Thus, while there is much general interest in sharing code between computer facilities as well as within them, automatic software distribution systems that do not allow for local control will most likely never be used when the source and destination machines are administered by different people.

An obvious alternative to central administration is to have receiving machines poll for new versions of files. Under such a scheme, the receiving machine would periodically check the status of each *master* file of interest and initiate updates when necessary. The problem with this approach is that unless checking file status and initiating updates are very easy, local administrators may often fail to do them. Furthermore, there must be very little overhead involved in checking the status of *master* versions since this check must be done once for every *copy* of the file during each polling period. However, it is interesting to note that while systems with distributed

initiation of updates can "simulate" a system with centralized control (assuming that the source machine can somehow execute a command on the destination machine,) the opposite is not true.

Another design assumption made in some automatic software distribution systems is the requirement that processes on both the source and destination machines operate in a synchronous manner. The `rdist` system makes this assumption; `asd` does not.

The Librarian/Subscriber Model of Updates

The system described in this paper, called `track`, was designed using distributed administration with periodic checking of file status and asynchronous communication. `Track` views the *master* versions of files as a passive repository. We describe this view as a *librarian/subscriber* model. The machine containing the master versions of files is called the *librarian* machine. (There can be more than one *librarian*. For ease of description we shall talk about only one.) Machines that make *copies* of files from the *librarian* are called the *subscriber* machines.

Clearly this is a server/client model. The different terminology is used to emphasize the functional similarity between our electronic *librarian* and its human counterpart. Typically, human librarians provide listings of available information and copies of information upon request. If human librarians followed the example of automatic software distribution systems with centralized control, they would march into one's offices without warning and insist that that a certain book is to be read immediately. We believe that such behavior on the part of either humans or machines is intolerable.

In keeping with our metaphor, our electronic *librarian* has two tasks : 1) to provide a listing of its files that are available to *subscribers*, as well as some information about the currentness of each file. `Track` uses the time of last modification as its measure of currentness. (Clearly, modification date is not the proper measure of currentness for things such as symbolic links and directories. For the time being, we shall only be discussing how `track` handles ordinary files. Special files as well as alternative measures of currentness will be discussed later in the section entitled "Nitty Gritty".) And 2) to give out *copies* of files to authorized *subscribers* upon request. Given such a cooperative *librarian*, a *subscriber* machine need only inspect the *librarian's* listing of available files and request new *copies* of files that are more current than its own.

At the core of `track's` operation is the *subscription list*. On the *librarian*, the *subscription list* describes the set of files to be made available to other machines. On the *subscriber* machine, the *subscription list* describes the set of files to be considered for potential update from the *librarian*. Often, the *subscription lists* are identical on *librarian* and *subscriber* machines. However, this is not always the case. As we shall see later, by editing local (*subscriber*) copies of the *subscription list*, local administrators can control updates on their machines.

A Simple Example

We will now step through an example of how updates take place. In order for an update to take place, the program named `track` must be run at some point on both the *librarian* and *subscriber*. While it is possible to invoke `track` by hand, it is more common for it to be invoked at regular intervals by the UNIX clock daemon (`/etc/cron`).

Figure 1 -- contents of the subscription list named "hourly"

```
#
#   files to be updated frequently
#
blob    : /etc/hosts   : : :
        : /etc/networks : : :
        : /etc/services : : :
```


At some point in time (as controlled by `/etc/cron`), the *librarian* machine ("blob") executes the command `track -w hourly`. Upon invocation with the `-w` flag (`-w` stands for "write") `track`:

- 1) reads the appropriate *subscription list*, ("hourly" -- see Figure 1). The *subscription list* specifies that the files named `/etc/hosts`, `/etc/networks`, and `/etc/services` are to be made available to *subscriber* machines. And,
 - 2) collects the modification date of each file listed and creates a *statfile*.^{*} By convention, the name of the *statfile* is created by combining the *subscription list*'s file name and the *librarian*'s machine name. In this case, the *statfile* is called "hourly.blob" (see Figure 2.)
- Statfiles* have one line for each object. Each line contains three pieces of information: 1) a single character describing the type of object being handled (in Figure 2, this is always "f" for "file"), 2) the name of the object, and 3) some measure of the *currentness* of the object (in Figure 2, this is always the modification date of the file expressed in seconds). For a complete description of the format of *statfiles* see the manual page `statfile(5)` in the appendix.

Figure 2 -- contents of the *statfile* named "hourly.blob"

```
f/etc/hosts 488058628
f/etc/networks 487994526
f/etc/services 482463012
```

At some later time (also determined by `/etc/cron`), the *subscriber* machine will execute the command `track hourly`. Upon invocation without the `-w` flag `track` will:

- 1) obtain a copy of the *librarian*'s *statfile*.
- 2) for each file in the *statfile*:
 - A) see if the file is included in its own *subscription list*
 - B) examine the modification date of its own version of the file.
- 3) request from the *librarian*, those files that are included in the *subscription list* and have newer modification dates. During the copying process, the *subscriber* must be careful to reset the date of last modification on the new local copies.[†]

It should be noted that we have not described how *subscribers* and *librarians* actually communicate with one another. It is our intention that `track` be able to work over many different communications media.

Some More Complicated Cases

Having described the basic case, we will now describe the layout of the *subscription list* in more detail. A more complete and formal description can be found in the manual page for `subscriptionlist(5)`.

The principal function of the *subscription list* is to define a set of files. In addition, the *subscription list* describes where *master* versions are to be found, and how they are to be copied. In other words, the *subscription list* defines what and how files are to be *tracked*.

A *subscription list* consists of a series of *entries*. Each *entry* consists of 6 fields. All fields, except the last, are terminated by a colon. The last field is terminated by a newline.

As in Figure 1, the first field contains the name of the *librarian* machine. The second field contains the name of the file to be *tracked*. To be more precise, the second field contains the file name to be used on the *librarian* (more on this later.) In the previous example, the second field always contained the name of an ordinary file. If the second field contains the name of a directory, that directory is handled as the root of a file subtree, i.e. all files in the subtree will be

^{*} *Statfiles* are named after the UNIX system call, `stat()`, that is used to collect file status and the modification date.

[†] If the modification date is not properly reset, then the next time that the *subscriber* examines the *librarian*'s *statfile*, it will erroneously conclude that the file is out of date and repeat the copying procedure.

tracked. The *entry*:

```
blob : /usr/lib : : :
```

causes all of the files in the subtree rooted at "/usr/lib" to be made available to other machines (on the *librarian*), or be considered as candidates for updating (on the *subscriber*).

Allowing subtrees to be treated as a unit has two advantages. First of all, it makes the *subscription list* shorter and much easier to read. Secondly, new files can be **tracked** merely by adding them to a directory that is already being **tracked**, without changing anyone's *subscription list*. It will be shown later how local administrators can intercept updates so as to avoid having new and possibly unwanted files magically appear.

However, there is a disadvantage to treating subtrees as a unit. Sometimes it is desirable to update most but not all of a subtree. For instance, almost everything in /etc can be shared between machines of the same architecture. However some files, such as "/etc/fstab" and "/etc/ttys"* usually have slight differences on all machines. In the *subscription list* the fifth field contains the names of files to be excluded from **tracking**. The *entry*:

```
blob : /etc : : fstab ttys :
```

can be read as "everything in /etc except /etc/fstab and /etc/ttys"†.

Given this design, *subscription list entries* define pruned subtrees of the filesystem. We have found this scheme to be concise yet easy to read. The *subscription list* that is used in our laboratory to update the 6000 files comprising 4.2BSD on VAX machines has 30 *entries* with 60 exceptions.

Command Execution

There is more to system administration than merely knowing when to copy files from one machine to another, a topic upon which any UNIX administrator will gladly expound. Often it is necessary to execute some command, or a series of commands after a file has been copied. For instance, there is a file on Berkeley 4.2BSD, named "/usr/lib/sendmail.cf," that describes the configuration of mail network. In order for any change to this file to be used by the mail system, the command **sendmail -bz** must be executed. An administrator can specify that this happen by putting the command in the sixth (and last) field of the *entry*. Thus, the *entry*:

```
blob : /usr/lib/sendmail.cf : : : : sendmail -bz
```

will cause the command **sendmail -bz** to be executed if and only if a new version of the file "sendmail.cf" is copied from the *librarian*. Or the local administrator can specify:

```
blob : /usr/lib/sendmail.cf : : : : \  
      echo new sendmail.cf has arrived | mail root
```

if he wishes to be notified of the update rather than having the installation happen automatically.‡
Or, the our local administrator may specify:

```
blob : /usr/lib/sendmail.cf : : : : \  
      echo got new sendmail.cf | mail root \  
      sendmail -bz
```

if he wishes to have it both ways.¶ Since the shell script resides in the *subscriber* machine's copy of the *subscription list*, local administrators retain full control over all details of the installation process.

* These files describe the layout of the file and terminal systems respectively.

† Actually, there are several other files in /etc that are usually excluded from **tracking**. For the sake of brevity, we show only two of them here.

‡ Note that the shell script may be more than one line in length. Newlines can be escaped with a backslash.

¶ These shell scripts have no effect on the *librarian* machine.

An important consequence of automatic command execution after a copy is the great ease with which one can maintain software on machines of different architectures.* In the past, an administrator has had the choice of either 1) cross-compiling the program and copying the executable binary to the second architecture or 2) transporting the code over to the second architecture and starting compilation by hand. **Track** provides a more straightforward solution. The entry:

```
blob : /usr/mystuff/src : : : \
      cd /usr/mystuff/src \
      make install
```

will cause the *subscriber* machine to update its source files from the *librarian* and then compile and install the new code using the makefile included in the source.[‡]

Changing Names As We Go

Track can map file names during the updating process. Often administrators want the new version of a file to have some name other than the exact name used on the *librarian* machine. For instance, this can be done whenever the administrator wishes to inspect the new versions of files before installation. In the *subscription list*, the fourth field is used to specify the name into which any new versions should be copied. Thus, the entry:

```
blob : /bin/sh : : /bin/Nsh : : \
      echo got new shell | mail root
```

will copy any new versions of the file named `/bin/sh` ON BLOB to the file named `/bin/Nsh` ON OUR LOCAL MACHINE and notify the administrator via mail.

Giving the new versions of a file a different filename is also useful when special installation procedures must be used. To continue with our example, great care must be taken when installing a new version of the shell. Because of the way UNIX operates, the old version of the shell must be moved aside before the new version can be moved into place. This can be done automatically by combining **track's** name mapping ability with a simple shell script. The entry:

```
blob : /bin/sh : : /bin/Nsh : : \
      mv /bin/sh /bin/Osh \
      mv /bin/Nsh /bin/sh
```

will copy any new version of the blob's file named `/bin/sh` to the locally named file `/bin/Nsh`. Then the local `/bin/sh` will be moved into `/bin/Osh` and `/bin/Nsh` will be moved into `/bin/sh`.

Track allows for a second type of name mapping. So far, we have shown how **track** can map names during the copying process. **Track** can use a different name mapping during the currentness checking process. In the previous examples, the name that a file has on the *librarian* is always the name that the file will eventually have on the local machine. The name `/bin/Nsh` was only temporary. This is not always the case. For instance, the unix kernel (`/vmunix`) is different on almost all machines. However, in our computer center, VAX kernel source code is kept on one machine where all the kernels are built. New kernels are then distributed to the other VAX machines. Gorp's kernel resides in `/sys/conf/GORP/vmunix` on blob and in `/vmunix` on gorp. Furthermore it would be inconvenient to use the name `/sys/conf/GORP/vmunix` on gorp itself, even for temporary storage. At best, gorp's administrator would have to "kluge up" the directories solely to allow automatic distribution of kernels. At worst, `/sys` may already contain something other than the current kernel source code. Rather than contorting the file system to

* Often, one wishes to use the same source code for a program that can be executed on machines that cannot use the same binary file.

‡ Clearly this technique has its hazards. Often "portable" code has unknown system dependencies. However, there have been few serious problems in our lab. If stderr from the make is automatically directed back to the maintainer of the code via mail, compile-time errors that result from unexpected machine dependencies can be quickly detected. Run-time errors are more difficult to handle.

meet **track**'s needs, the third field of the *subscription list entry* contains the name of the file ON THE LOCAL MACHINE to be used when checking currentness. Thus, the *subscription list entry*:

```
blob : /sys/conf/GORP/vmunix : /vmunix : /nvmunix : : \  
      echo new kernel has arrived | mail root
```

will cause **track** to compare blob's file named /sys/conf/GORP/vmunix with the local file named /vmunix and if the versions differ, the new kernel will be copied to yet a third location, namely /nvmunix (with appropriate notification made to the administrator). Using a slightly different shell script:

```
blob : /sys/conf/GORP/vmunix : /vmunix : /nvmunix : : \  
      mv /vmunix /ovmunix \  
      mv /nvmunix /vmunix \  
      shutdown -r now
```

a *subscriber* can copy and install a new kernel without human intervention.* The previous example opens up the possibility of having virtually "hands-off" system administration. While this level of automation is not necessary at the moment, such capability may be very useful when computer centers contain hundreds rather than dozens of CPU's.

Nitty Gritty

The following section includes a discussion of some of the finer points of the updating process and design decisions made in **track**. It is intended to satisfy the curiosity of UNIX "guru's." As a result, it may not be of interest to all readers.

In addition to handling ordinary files, **track** has been designed to handle directories and symbolic links. No attempt was made to handle character or block devices. Some special difficulties arise when handling directories and symbolic links. In particular, how does one determine the currentness of a directory or symbolic link? Clearly modification date is not the right measure. For the purposes of discussion, we shall say that **track** *evaluates* the *currentness* of each file. The *evaluation* of an ordinary file is its modification date. Directories *evaluate* to their mode bits and owner's uid and group's gid. Symbolic links *evaluate* to the name of the file towards which the link is pointing.

There are other ways to *evaluate* an ordinary file. Another possible *evaluation* might include the mode bits and owner of the file (similar to the way in which directories are evaluated). Another approach would be to have a check sum calculated for each file. Each of these makes a different tradeoff between certainty that the file is correct and time needed to *evaluate* the file.

Like any other program **track** sometimes has trouble doing its job. However, unlike most other programs, **track** may be executing on a machine that is far away from the person maintaining the files that **track** is handling. Therefore, if **track** is invoked with the -m flag, stderr from **track** (and any shell scripts spawned by **track**) is sent to the administrator via mail. The default recipient of mail is "root." This can be changed by specifying a different recipient after the -m flag.[‡]

What's Ahead

The **track** system while presently useful, is still undergoing development. The shape of future work will most likely be driven by needs that appear as the system is used more widely. At present, the system handles updates to about 20 machines running Berkeley 4.2BSD UNIX connected via an ethernet. Future work will allow **track** to operate via a uucp link between

* We do not recommend that this procedure be employed regularly on heavily used systems. However, we believe that it is an excellent example of the power derived by combining **track**'s two styles of name mapping with automatic command execution.

‡ When **track** is invoked without the -m flag, stderr will be directed to the tty of invocation. Thus, if **track** is invoked by **cron** without the -m flag, stderr is lost.

machines. In addition, we plan to port **track** to other flavors of UNIX (i.e. System V and Research Version 8).

Given operation via **uucp**, **track** should be able to make the life of system administrators much easier. **Track** can be used for distributing bug fixes as well as new releases of software packages or even UNIX itself. It is our hope that **track** will allow much more sharing of code than has been possible in the recent past.

Reference

- [1] Koenig, A. Automatic Software Distribution. *Proceedings of the USENIX Summer Conference*. pp. 312-322. Salt Lake City, Utah. June 12-22, 1984.

NAME

track -- maintain files between machines

SYNOPSIS

track [- drnpqmfw] *subscriptionlist*

DESCRIPTION

subscriptionlist contains a list of files to be maintained across machines. **track** supports a *librarian/subscriber* model of file update. If **track** is invoked with the **-w** option (i.e. as the *librarian* machine), then the *subscriptionlist* is used to create a *statfile* containing status information about the files that are available to other machines. If **track** is invoked without **-w** (i.e. as the *subscriber* machine), then the *subscriptionlist* is used to decide what files are to be considered for possible updating, and what machine is to be used as the source of updated files.

Other options are:

- n** Do nothing, just produce a list of files that need updating (disabled by **-w**)
- p** Parse only and produce a verbose description of the fields in the *subscriptionlist*
- d**<dirname>
change the local working directory (i.e. the directory on the local machine) to *dirname*. A working directory is the place where **track** will look for *statfiles* and *subscriptionlists*, as well as the place where **track** will attempt to create scratch files. **Track** must have permission to write in its local working directory. The default is */usr/track*.
- r**<dirname>
change the remote working directory (i.e. the directory on the librarian machine) to *dirname*. The remote working directory will be the same as the local working directory unless the **-r** flag is used.
- u** Unconditionally copy files (i.e. older files will overwrite newer files)
- f** Force the update. Don't be gentle about what is destroyed. Normally **track** is very cautious about files being destroyed by an update.
- m**[recipient]
Mail stdout and stderr to the recipient. The default recipient is root.
- q** Be quiet, i.e. don't complain about non-fatal errors. The most common non-fatal error occurs when the *librarian* and/or network are unreachable.
- w** write out a *statfile*. This flag is always used by the *librarian* machine and never used by the *subscriber* machine(s). In fact, it is the use of the **-w** flag that differentiates *librarian* and *subscriber* machines.

EXAMPLES

track -p daily

Parse the *subscriptionlist* named "daily" and print a verbose listing of the contents. This command is very helpful when trying to debug freshly modified *subscriptionlists*.

track -w hourly

Act as the *librarian* and output a *statfile* for the *subscriptionlist* named "hourly". The *statfile*'s name is a combination of the machine name and the *subscriptionlist* name. Thus, if the machine on which this command is executed is named "blob" the *statfile* will be named "hourly.blob".

track -w -q hourly

Same as above, but non-fatal errors will be ignored.

track -w -q -m hourly

Same as above, except that error message will be mailed to root rather than sent to stderr. These are the arguments most often given when **track** is executed by **cron(8)** on the *librarian* machine.

track -w -q -mfoo!bar

Same as above, except that error messages will be mailed to "foo!bar" rather than "root"

track daily

Act as a *subscriber* machine and get the updates from whatever *librarian* machine is specified in the *subscriptionlist* named "daily".

track -n daily

Act as a *subscriber*, but do not perform any updates. Instead, print a listing of the updates that should be made. This is analogous to "make -n". Invoking **track** with the **-n** flag is a safe way to preview what will happen before **track** is turned loose on a new machine.

track -u daily

Act as a *subscriber*, except update files from the *librarian* even if the versions on this machine are newer than the versions on the *librarian*.

Without the **-u** flag, **track** will not copy an older file on top of a newer one.

track -u -m daily

Same as the above, except mail error messages to "root" rather than sending them to stderr.

track -u -m -q daily

Same as the above, except ignore non-fatal errors. These are the arguments most often given when **track** is executed by **cron(8)** on a *subscriber* machine.

FILES

/usr/track/*

/usr/track/ *subscriptionlist_name*

/usr/track/ *subscriptionlist_name* . *librarian_machine_name*

SEE ALSO

subscriptionlist(5), statfile(5), cron(8)

When Network File Systems Aren't Enough: Automatic File Distribution Revisited
by Daniel Nachbar

NAME

statfile – list type, name, and *currentness* of files for **track**

DESCRIPTION

A *statfile* is a listing of objects to be handled by the **track** system. Objects are either ordinary files, directories, or symbolic links. *Statfiles* contain one line for each object. The format of the lines is as follows:

The first character on the line describes the type of object being handled. This character must be either "f" for ordinary files, "d" for directories, or "l" for symbolic links.

Immediately following the first character (i.e. with NO white space separation), is the full path name of the object.

Whitespace -- space(s) and/or tab(s) -- must follow the pathname. The last item on the line is a string that describes the *currentness* of the object. For ordinary files, this string is the modification date of the file expressed in number of seconds since the epoch. For directories, this string consists of the uid of the directory's owner, followed by a period ("."), then the gid of the directory's group, another period, and finally the mode bits of the directory (in octal). For symbolic links, the string consists of the name of the file towards which the link is pointing.

EXAMPLES

```
d/bin 0.0.40775
f/bin/df 476233823
l/usr/ucb/newaliases /usr/lib/sendmail
```

The above three lines tell **track** that the directory `"/bin"` is available and it is owned by "root" as well as group "root" and has modes "40775" (Normally written as "drwxrwxr-x"). Similarly, the ordinary file named `"/bin/df"` is available to *subscribers* and its modification date is "476233823 seconds since the epoch" (Normally written as 6:10:23 on February 2, 1985). The symbolic link `"/usr/ucb/newaliases"` is also available, and is pointing to `"/usr/lib/sendmail"`.

FILES

`/usr/track/ subscriptionlist_name . librarian_machine_name`

BUGS

Mode bits as well as owner and group should probably be included in the *currentness* string for ordinary files. The uid and gid should be replaced by owner and group name. Also, the "40" that preceeds the other mode bits for directories is redundant with the initial type specifier. One of them should be removed.

SEE ALSO

`track(1)`, `subscriptionlist(5)`

When Network File Systems Aren't Enough: Automatic File Distribution Revisited
by Daniel Nachbar

NAME

subscriptionlist – describe files to be updated by **track**

DESCRIPTION

A **subscriptionlist** defines a set of files to be considered for updating by the **track** system. A **subscriptionlist** contains a set of *entries*. Each *entry* has 6 fields. The fields are (in order) :

- 1) The name of the *librarian* machine (i.e the machine from which updates are to be made.)
- 2) Name of the source file on the *librarian*.
- 3) Name of the file on the *subscriber* machine to be used for determining if an update is necessary.
- 4) Name of the destination file on the *subscriber* machine.
- 5) A list of the files to be excluded from updating.
- 6) A shell script to be executed whenever an update is made.

Fields are separated by a colon. The last field (and thus the entry) is terminated with a newline. White space is ignored in all fields except 5) and 6). In 5), whitespace is used to separate entries. In 6), all white space is preserved and passed to `/bin/sh`. Newlines in 6) can be escaped with a backslash. The backslash will be removed before the script is sent to the shell. Any characters after a hashmark, "# ", up to the end of line will be considered to be comments and will be ignored.

If the name of a directory is used in any field, **track** treats the directory as being the root of a subtree. If the name of a symbolic link is used in any field, **track** will handle the symbolic link itself and will NOT follow the link.

The action taken upon encountering a blank field differs for each field. The actions are:

- 1) Use the the name of the *librarian* from the previous *entry*. Leaving this field blank in the first *entry* will cause a unrecoverable error.
- 2) Unrecoverable error -- this field must be filled in each *entry*.
- 3) Use the name given in 2) of this entry.
- 4) Use the name given in 3) of this entry. If 3) is itself empty, use the name given in 2).
- 5) No effect
- 6) No effect

EXAMPLES

```
xyz123 : /etc/host :::
```

This entry defines the machine named "xyz123" as the *librarian* for the file named "/etc/host".

```
xyz123 : /etc/host :::
```

```
: /etc/services :::
```

These entries define "xyz123" as the *librarian* for both "/etc/host" and "/etc/services".

```
lassie : /usr/lib :::
```

This entry defines the machine named "lassie" as the *librarian* for the entire subtree rooted at "/usr/lib".

```
lassie : /usr/lib :: sendmail.cf uucp :
```

This entry will use "lassie" as the *librarian* for all files in the at "/usr/lib" exce for "/usr/lib/sendmail.cf" and all files in the subtree rooted at "/usr/lib/uucp

```
xyz123 : /usr/lib/aliases : : : newaliases
```

This entry will use "xyz123" as the *librarian* for the file "/usr/lib/aliases" and will execute the command "newaliases" if and only if a new version is successfully copied.

```
xyz123 : /usr/lib/aliases : : : newaliases \
echo aliases file has been updated | mail root
```

This entry works exactly like the previous example and in addition sends mail to the administrator.

```
xyz123 : /usr/lib/aliases : : : newaliases \
echo 'date' /usr/lib/aliases >> /usr/adm/tracklog
```

This entry works exactly like the previous example except that rather than sending mail, a log entry containing the date and filename is appended to the file named "/usr/adm/tracklog".

```
xyz123 : /usr/lib/aliases : /tmp/aliases : : \
echo new aliases file has arrived | mail root
```

This entry copies any new versions of the file named "/usr/lib/aliases" on "xyz123" to a locally named file "/tmp/aliases" and sends mail to the administrator.

```
xyz123 : /bin/sh : /bin/nsh : : \
mv /bin/sh /bin/osh \
mv /bin/nsh /bin/sh \
echo new shell installed | mail root
```

This entry copies any new versions of the file named "/bin/sh" on the *librarian* (xyz123) to a locally named file "/bin/nsh", then saves "/bin/sh" in "/bin/osh", then moves the the new version into "/bin/sh".

```
xyz123 : /usr/sys/GORP/vmunix : /vmunix : /nvmunix : : \
mv /vmunix /ovmunix \
mv /nvmunix /vmunix \
shutdown -r now
```

This entry causes **track** to compare the locally named file "/vmunix" with the *librarian's* (xyz123's) file named "/usr/sys/GORP/vmunix". If the versions differ, the new version will be copied into the locally named file "/nvmunix" and the shell script will be executed to install and reboot with the new kernel.

FILES

/usr/track/ *subscriptionlist_name*

BUGS

A comment after the last entry will cause a fatal parsing error.

SEE ALSO

track(1)

statfile(5)

When Network File Systems Aren't Enough: Automatic File Distribution Revisited
by Daniel Nachbar

Experiences Implementing BIND, A Distributed Name Server for the DARPA Internet

James M. Bloom

Kevin J. Dunlap*

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley CA 94720

1. Introduction

The Berkeley Internet Name Domain (BIND) Server implements the DARPA Internet name server for the UNIX[†] operating system. A name server is a network service that enables clients to name resources or objects and share this information with other objects in the network. This in effect is a distributed data base system for objects in a computer network. BIND is fully intergrated into 4.3BSD network programs for use in storing and retrieving host names and address. This paper describes our experiences implementing BIND for the DARPA Internet, including the design decisions that were made, the problems that were encountered and directions for future development.

1.1. The Name Service

The basic function of the name server is to provide information about network objects by answering queries. The specifications for this name server are defined in several Internet Requests For Comment, RFC: RFC882, RFC883, RFC973 and RFC974.

The advantage of using a name server over the host table lookup for host name resolution is to avoid the need for a single centralized clearing house for all names. The authority for this information can be delegated to the different organizations on the network responsible for it.

The host table lookup routines require that the master file for the entire network be maintained at a central location by a few people. This works fine for small networks where there are only a few machines and the different organizations responsible for them cooperate. But this does not work well for large networks where machines cross organizational boundaries.

* This author is an employee of Digital Equipment Corporation's Ultrix Engineering Advanced Development Group and is on loan to CSRG. Ultrix is a trademark of Digital Equipment Corporation.

†UNIX is a Trademark of AT&T Bell Laboratories

With the name server, the network can be broken into a hierarchy of domains. The name space is organized as a tree according to organizational or administrative boundaries. Each node, called a *domain*, is given a label, and the name of the domain is the concatenation of all the labels of the domains from the root to the current domain, listed from right to left separated by dots. A label need only be unique within its domain. The whole space is partitioned into several areas called *zones*, each starting at a domain and extending down to the leaf domains or to domains where other zones start. Zones usually represent administrative boundaries. An example of a host address for a host at the University of California, Berkeley would look as follows:

monet.Berkeley.EDU

The top level domain for educational organizations is EDU; Berkeley is a subdomain of EDU and monet is the name of the host.

1.2. Overview of BIND

BIND is comprised of two parts as illustrated in figure 1. One is the user interface called the *resolver* which consists of a group of routines that reside in the C library */lib/libc.a*. The *resolver* is comprised of a few routines that build query packets and exchange them with name servers. The larger part is the actual server called *named*. This is a daemon that runs in the background and services queries on a specific network port.

The most common configuration of a BIND system is a BIND server running on the local machine that may or may not have authoritative information about the local and other domains. The server maintains a cache of responses received in order to improve performance when common queries are asked. As an example, consider a client of some network service that wants to connect to its corresponding server. The first thing it must do is find out the network address of the server. The client calls a resolver stub through *gethostbyname()* that builds a query for the data base. This query is then sent to *named* on the local machine. If the server has the information in its cache or data base, the answer is returned immediately, otherwise the server forwards the query to the server for the domain that it is trying to find. The response to this query may either be the answer or may refer the server to another

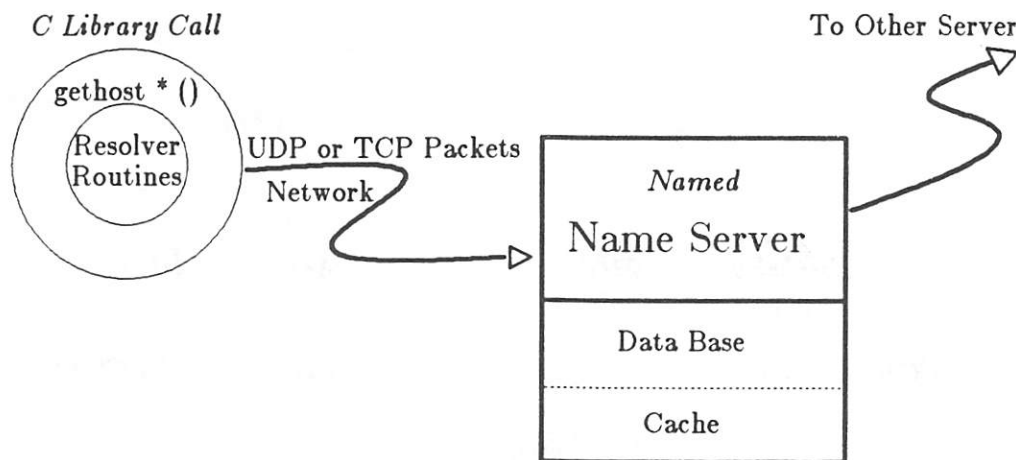


Figure 1: Structure of BIND

server with more complete information. After a few queries, usually one or two, the answer is returned.

Figure 2 shows an example domain tree. Each of the circles represents a server, or group of servers, responsible for the domain labeled on the circle. Suppose a network application on decvax.DEC.COM wants to find the address of fignewton.LCS.MIT.EDU. The application program would send a query packet to the local name server. The local name server would in turn send a packet to the root name server, that would respond saying to try the EDU name server. Then a query to the EDU server would get an answer saying to try the MIT name server. The MIT name server would respond saying to try the LCS name server. The final query is then sent to LCS.MIT.EDU's server that would then respond with fignewton's address. This is purely a theoretical example. The EDU, COM and root name servers are currently the same name server. Name servers also have a cache that allows them to retain information for a designated amount of time.

2. Design Decisions

2.1. Iterative Resolution and Cache Placement

Two issues that had to be approached together were where to place a data cache and how iterative queries should be handled. Since a query may just yield a referral to another name server, we must be able to handle iterative queries. The cache interacts with the iterative processing in that the cache may contain some, but not all the information needed to process the query.

To improve performance, a cache of query responses was needed, as a query over the internet may take several seconds. There were several options as to where the data caching should be done. The choices were in the resolver, in the server, or some combination of the two. Different performance improvements could be gained

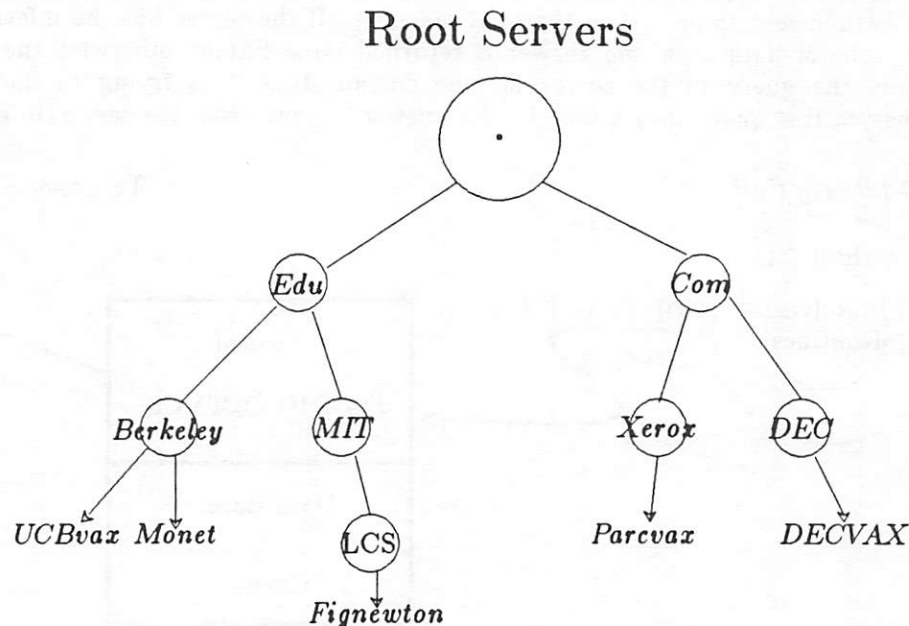


Figure 2: Sample Internet Domain Tree

from each choice.

The iteration required for queries could also be done in more than one location. Either the resolver could do the iteration, or the name server could handle it. This decision has some effect on where the cache should be placed since the iteration should use the cache for assistance.

Placing the cache with the resolver would give individual programs more control over their cache. A set of operations could have been designed to allow maintenance of the cache. This would allow programs that did not care to use the cache to disable it. The program might have been able to flush sections of the cache as well. The major problems with placing the cache in the resolver are that it adds more code to every program using the name server and most programs only make one or two queries to the name server during their lifetime. Because of the few queries here, the cache is useless. Also, other programs could not share the cache.

If the cache is located with the resolver, iterative query processing would need to be done by the resolver. If not, the name server would have to duplicate all the work every time it got a query for a machine in a domain requested a short time before. The cache would save pointers to the name servers to query or possibly the answer itself.

Splitting the cache between the resolver and the name server is possible. This would work best with the resolver doing iterative queries. Here the resolver would cache information about the location of name servers for other domains and the name server would cache other information. The drawback to this scheme is having duplicate code for managing the caches. The name server would also need to get the information from the resolver to store in its cache. This causes additional packets in every exchange. This scheme also suffers from the problem that most processes make few queries.

Finally, the cache could be placed in the name server. This scheme has the advantage that all processes could take advantage of the cache. If the iteration is performed by the name server, this helps the code doing the iteration by having as much information as possible to use. This may reduce the number of packets that are exchanged in resolving some queries. Putting the cache in the name server also reduces the size of every program using the name server since they do not need code for maintaining the cache. The shared cache should also lead to fewer queries over the internet since all programs will be sharing the results.

The machine wide availability of the name server lead us to place the cache in the name server. In light of this, the best place for the iteration was determined to be in the name server as well. This leaves the resolver as small as possible without affecting the functionality. Our method of handling the resolver differs considerably from the one described by Paul Mockapetris in the *Request for Comment* describing the name server implementation [RFC883]. Paul's design describes a system developed on a different architecture. A host name lookup is similar to a system call. This implies that the resolver is shared by everyone and placing the cache in the resolver gets the same performance. The resolver also does the iteration here as well and the name server just provides information for the supported zones.

2.2. Address to Name Translation

Mapping internet addresses to host names is a desirable feature that had been provided by the host table look up routines. Most users are shielded from the complications of internet addresses by the inverse mapping. This same functionality is desired in the name server as well. Two different methods have been tried for doing the address to host name translation. The first method was looking up the

address in the cache from previous request for internet addresses. This was unreliable as the common first question about a host was to find the host name from the address instead of the reverse. As a result, the information usually was not found in the cache. This method did not provide any way to expand the search beyond the local cache.

The second method is to have a hierarchical domain comprised of internet addresses. A name in this domain is comprised of an ASCII string in inverse order from the internet address followed by a standard domain name specifying the inverse query. For example, one might request the address of 128.95.253.18 by issuing a query for "18.253.95.128.in-addr.arpa". Since a site might be assigned the network address 128.95, the reverse ordering of the address allows for the hierarchical delegation of authority.

2.3. Appending Default Domain Name

With domain style addressing a fully qualified name is one that has at least one "." in it. When people want to specify a machine in their local domain they do not want to use the fully qualified name to send mail or copy files between them. With BIND, we made a design decision that clearly was not covered in the RFC's. When a user specifies a name that does not have a "." in it the name server will tack on the default domain name for the current domain. If a name server from another domain queries for an unqualified name, they might get an answer that is not what they wanted. It is clear in the RFC that queries should be for fully qualified domain names. If a server from another domain does not use a fully qualified name in a query, the response always should be *Host Unknown*. Our feeling is that if you do not use a fully qualified name in a query outside your domain, you can afford to receive incorrect information. This makes life easier for people by not having to specify a full domain for hosts in their local domain.

2.4. Garbage Collection

While developing BIND, we had to figure out how to handle old entries in the data base. All entries that are not in a server's zones have a *time to live* (ttl) field. After the server has been running for a while, it collects entries that are no longer valid based on their ttl field. We had several choices of how to remove the invalid entries. First, we could write a routine that walked through the data base and looked for expired entries. This would require that the server become unavailable for answering queries while doing the garbage collection. We did not consider this acceptable on a busy machine since a large backlog of requests could build up during the time required.

Another scheme for handling the garbage collection would have been to do it when the server was idle. On a busy machine, there may not be enough time to cover a reasonable percentage of the data base before being interrupted to handle another query. Since working on a query may lead to the data base being modified, saving of state would be difficult. It might have been possible to leave parts of the data base locked while doing the garbage collection, but this entails writing a large amount of code to handle the possibility of the query trying to access the part of the data base being cleaned. To run the clean up straight through gets back to all the problems mentioned above.

We finally decided that we would almost ignore the problem. The only time that old data is removed is when we get another query for it. This requires only minimal additional time while looking up information since only a handful of entries will be removed. All the entries removed would be in the normal look up path

through the data base. No locking is required.

2.5. Query Status Information

One feature that was required but not anticipated originally was the need to develop an interface for telling the program why the query failed. This was especially useful for the mail system that wanted to know if the failure was temporary or the request was invalid. There were existing interfaces for the requests and these only provided for returning success or failure. To solve this problem, a new variable with semantics similar to the Unix system call return status *errno* was introduced. This allowed programs that wanted more information to be able to retrieve it. Unfortunately, this is not a good interface for error returns, but currently we want to maintain compatibility.

3. Problems Encountered along the way

3.1. Improving Reliability

Reliability turned out to be a problem with the original design. In a distributed environment the server cannot always rely on the accuracy of the information it receives. A server may be put on the network that accidentally or maliciously distributes inaccurate information. Over the last year there have been several instances of servers stating that they know of a server that is authoritative for the root domain when in reality it was not. Other servers on the network would trust this information and tell other servers. The result is that the server listed as knowing something starts getting queries for things it knows nothing about. This wastes time for everyone and sometimes results in the spreading of more inaccurate information. Work still is needed on validating received information.

3.2. Educating User Community

Educating the Berkeley user community on the new form of addressing turned out to be a major task. When you change the way people address a piece of mail they get really upset if the address they used yesterday does not work today. This is analogous to the US Postal Service changing everyone's street address and then returning everyone's letters that do not use the new address. Gradually over five months we brought the domain mailing addressing into effect on campus. Every couple of weeks there would be a message posted on campus explaining what was happening and how to convert old style addresses to new domain style addresses. While the entire campus was being converted over to domain addressing the masses let it be known that they were not happy about this new style of addressing. Many other problems in the internet at the same time were being blamed on the name server. The name server had the same problems as all other applications of the network, but it changed the error messages that were received and returned them earlier. After a while people got accustomed to the new addressing and the number of complaints decreased.

A common complaint heard now is that there is no easy way to find the name of a machine in another domain. People were able to look in the host table to find a host name. The name server does not provide an interface for looking up arbitrary names. One problem is that knowing the name of an organization is not enough to find out its domain name. A user must figure out the name used by an institution and the domain in which it resides. For example, University of California at Davis is planning on using Davis.EDU while the University of Maryland at College Park is using UMD.EDU. Some tools are being developed to assist users in looking for host

names using the name server, but more work is still needed.

3.3. Problems with Single thread TCP connections

The original implementation of *named* only allowed for a single TCP connection for queries. When this connection was established no other queries could be serviced until the query on the TCP port was completed. This caused many problems in query processing. UDP queries would backup in the UDP receive buffer. If more than one query wanted to use TCP at the same time the second one would have to wait. If the server needed to query another name server to resolve the query, *named* sat idle waiting until the other name server responded and *named* could return the answer. *Named* can now service multiple TCP connections at the same time as illustrated in figure 3. The number of connections is limited by the number of available file descriptors to a process. There is now a linked list structure of open TCP connections. This structure contains a buffer for the incoming query, the file descriptor, a time stamp of the last time the connection was active, and the size of the pending query. All I/O on the connection is non-blocking. The connection is only serviced when it becomes active. This way multiple TCP connections and UDP queries can be serviced simultaneously.

If a query comes in via UDP, *named* checks to see if it has the information in its data base. If it does, then the response is returned to the requester; if not, and it has to query another server, it sends a query to the other server and then time stamps it and places the query on the *Query Retry Queue*. After a retry timeout, the query is resent.

For TCP queries the same process takes place as UDP. The only difference is that *named* manages a queue of TCP connections. If *named* runs out of file descriptors then it walks the TCP connection queue looking for any connections that have not been active for 15 minutes and closes them off. As it is walking the queue it is also looking for the oldest connection over 5 minutes. If after walking the queue, *named* is still out of file descriptors this connection is closed.

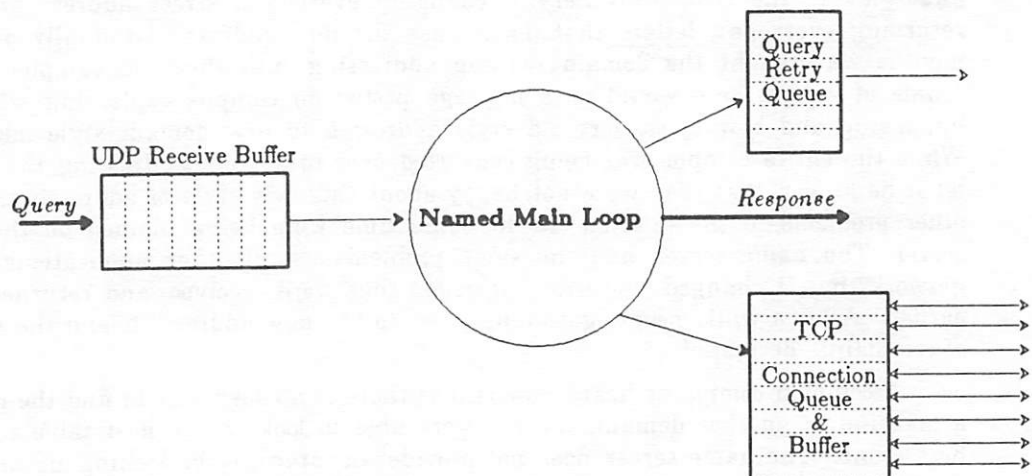


Figure 3: Front end of Named

4. Outline of Future Work for BIND

BIND is usable and is the default way of doing host name and address resolution for 4.3BSD. There are several areas where BIND can be enhanced for ease of use and expanded functionality. These include better ways for starting up a server, improved query processing and new functions for changing data in the server.

4.1. Changes in Starting up a Server

4.1.1. Hiding the initial cache data

Currently the data for locating the root servers is loaded from a file with the time to live field set to several years. The idea is that the data would not expire, however the bad effect is that when the server passes this information to other servers they too retain this data for years. This data should be loaded and stored elsewhere and used to find the root name servers when all other information has timed out.

4.1.2. Dumping and loading selective zones

Currently the name server cannot dump and load a selective zone; it can only dump the whole data base. For future development it will be necessary to dump portions of the data base for storing information received over the network. If the server cannot load the information over the network the next time it is restarted, the file can be used as a backup until it can successfully get the information from the network.

4.1.3. Elimination of Startup Files

The name server needs at least one file for initial bootstrapping. It would be desirable to develop some protocol that allowed a server to find out the small amount of information that it needs to run properly. This protocol will lead to fewer files having to be maintained for the name server. One possibility is for the server to broadcast a request for a master on the local network, but other possibilities should be examined as well.

4.2. Query Processing

4.2.1. Distributed Server Loading

Currently the name server tries a list of servers in the order given to it. This tends to overload the first server on the list with all the queries. Methods of balancing the load between equivalent servers that even the load and improve response time should be examined. A round robin approach ordered by measurements of previous response times from the servers is one option to be investigated.

4.2.2. Host address sorting

Currently the name server returns host addresses in the order that they are stored in the data base. This is not always the best order for a host with multiple network interfaces. The addresses should be sorted in an order best suited to the requester. To determine the proper order, some knowledge of network topology is required, but even routing protocols may not be able to provide enough information to get this correct.

4.3. Data Base Update Functions

As more information is stored in the name server's data base, people are going to want to change information. Currently to make changes to the data, the name server must reload the whole data base and purge all cached data. During the time required to load the data base, the name server cannot respond to queries. The standard data base functions of insert, delete, and modify a record are needed. These functions should have minimal effect on other functions.

4.3.1. User Update Mechanism and Record Protection

A function of *named* that is not well integrated into the rest of the system, is information about mail delivery, mail routing, and mailing lists. Users will want to change this information for themselves when the system starts using these functions. For this to happen there needs to be a protection and authentication scheme, so that a user cannot change someone else's data but is permitted to change their own.

4.3.2. Master Servers updating each other

A method of keeping track of changes to the data base is needed. This will allow equivalent masters to exchange only the changes and not the entire data base. It is assumed that only a few records will change each hour and the cost of the update processing can be minimized. Work also needs to be done on authentication, to validate the server providing the update and to avoid security problems.

ACKNOWLEDGEMENTS

Many thanks to the users at U.C. Berkeley for falling into many of the holes involved with integrating BIND into the system so that others would be spared the trauma. Thanks are extended to Digital Equipment Corporation for permitting Kevin to spend most of his time on this project.

This work was sponsored by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871 monitored by the Naval Electronics Systems Command under contract No. N00039-84-C-0089. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency, of the US Government, or of Digital Equipment Corporation.

REFERENCES

- [Birrell] Birrell, A. D., Levin, R., Needham, R. M., and Schroeder, M.D., "Grapevine: An Exercise in Distributed Computing." In *Comm. A.C.M.* 25, 4:260-274 April 1982.
- [Dunlap] Dunlap, K. J., "Name Server Operations Guide for BIND" *Unix System Manager's Manual, SMM-11*. 4.3 Berkeley Software Distribution, Virtual VAX-11 Version. University of California, Berkeley. April 1986.
- [RFC819] Su, Z. Postel, J., "The Domain Naming Convention for Internet User Applications." *Internet Request For Comment 819* Network Information Center, SRI International, Menlo Park, California. August 1982.
- [RFC882] Mockapetris, P., "Domain Names - Concept and Facilities." *Internet Request For Comment 882* Network Information Center, SRI International, Menlo Park, California. November 1983.
- [RFC883] Mockapetris, P., "Domain Names - Implementation and Specification." *Internet Request For Comment 883* Network Information Center, SRI International, Menlo Park, California. November 1983.
- [RFC973] Mockapetris, P., "Domain System Changes and Observations." *Internet Request For Comment 973* Network Information Center, SRI International, Menlo Park, California. February 1986.
- [RFC974] Partridge, C., "Mail Routing and The Domain System." *Internet Request For Comment 974* Network Information Center, SRI International, Menlo Park, California. February 1986.
- [Terry] Terry, D. B., Painter, M., Riggie, D. W., and Zhou, S., *The Berkeley Internet Name Domain Server*. Proceedings USENIX Summer Conference, Salt Lake City, Utah. June 1984, pages 23-31.
- [Zhou] Zhou, S., *The Design and Implementation of the Berkeley Internet Name Domain (BIND) Servers*. UCB/CSD 84/177. University of California, Berkeley, Computer Science Division. May 1984.

Network Performance and Management with 4.3BSD and IP/TCP

*Michael J. Karels
Marshall Kirk McKusick*

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

The 4.3BSD (Berkeley Software Distribution) release of UNIX[†] includes a number of changes to the IP/TCP support originally released in 4.2BSD. Many of these changes are bug fixes or modifications that increase tolerance of errors and other unusual conditions. There are several changes that specifically address performance. Some of these changes are designed to increase throughput or minimize delay for an individual connection. Other changes attempt to match the hosts to the overall network, minimizing congestion in the network or in the gateways between the hosts. This paper describes the changes in both areas. It also describes some experiences in managing local networks that include 4.3BSD and other hosts using Internet and other protocols. A discussion of the considerations in network layout, routing, use of subnets, and gateway usage is provided. Some of the potential pitfalls are discussed, with examples of disasters that may occur if improper choices are made.

1. Introduction

Considerable effort was expended during the development of the IP/TCP [Postel81a][Postel81b] network protocol implementations in 4.2BSD [Leffler83] to improve performance. The performance optimization was targeted primarily to high-speed local area networks such as Ethernet[‡]. In that environment the protocols were able to use fairly large data segments. With large segments, a smaller number of packets were required for a data transfer. A packet size of 1024 bytes allowed the use of page cluster-sized buffers, avoiding copy operations by remapping pages. The variable-sized headers used by the Internet protocols made it difficult to align receive buffers so that data would be page-aligned, requiring copy operations at the receiving host. This led to the use of trailer encapsulations [Leffler84a], which use a link-layer convention that places variable-sized headers at a known location after the data. This allowed the receiving host to anticipate the data location, aligning it in the receive buffer at the beginning of a page-sized buffer. However, trailer packets caused compatibility problems with non-4.2BSD hosts that do not understand them.

Although the network performance of 4.2BSD was reasonably good on Ethernets, some of the changes that improved local-net throughput were less optimal for long-haul networks. The maximum segment size of 1024 bytes worked poorly on a network such as the ARPANET, which has a maximum message size of 1007 bytes. This paper describes changes made since the release of 4.2BSD that

[†] UNIX is a trademark of Bell Laboratories.

[‡] Ethernet is a trademark of the Xerox Corporation.

VAX and UNIBUS are a trademarks of Digital Equipment Corporation.

address these problems and other performance considerations. Other work in this area has been described previously [Leffler84b][Walsh84].

2. Performance Optimization

Although 4.2BSD was already reasonably well tuned for fast local networks, additional performance was found to be possible by increasing the buffer sizes on both sending and receiving hosts for TCP stream connections. The default buffer size was increased from 2048 to 4096 bytes. With 1024-byte segments, the sending host was able to send 4 packets rather than 2 before requiring a window update. This change improves the effectiveness of delayed acknowledgment in reducing the number of acknowledgement-only packets; the delayed acknowledgement strategy is to withhold acknowledgements until a window update would uncover at least 35% of the window. With 1K packets and 2K buffers, no delay was possible, but with the larger buffer sizes, every other acknowledgement can be eliminated, as TCP acknowledgements are cumulative. As acknowledgement processing is a substantial fraction of the CPU load on the sending side, a 50% reduction in acknowledgement-only packets produces a substantial reduction in CPU load on the sender. This change also increases the amount of parallel activity possible, with both sender and receiver active. For high-bandwidth needs such as remote backups, a new socket option was added to allow a user process to increase the buffer size allocated to it. Remote *dump* and the remote tape server *rmt* use this option to set the network buffer to the same size as a tape block.

The network buffer handling routines are optimized for 1024-byte buffers, and the user send routine attempts to place data in buffers of that size. In 4.2BSD, this routine would use large buffers only when there was at least a kilobyte of data to send and there was sufficient space for the full kilobyte in the output queue. The send routine in 4.3BSD will use large buffers if they would be at least half full. Also, if there is insufficient space for a full buffer in the output queue, but there is outstanding data already, the send routine will wait for the queue to drain rather than splitting the send into smaller pieces. The low-level UNIBUS interface support routines that receive packets from the network and place them in network buffers will also use large buffers if they will be at least half full.

4.2BSD offered a maximum receive segment size of 1024 for all connections, and accepted such offers whenever made. However, that size was especially poor for the ARPANET and other 1822-based IMP (PSN) networks where the maximum packet size is 1007 bytes. This was compounded by a bug in the LH/DH driver that did not allow space for an end-of-message bit in the receive buffer, and thus maximum size packets that were received were split across buffers. This, in turn, aggravated a hardware problem causing small packets following a segmented packet to be concatenated with the previous packet. The result of this set of conditions was that performance across the ARPANET was sometimes abominably slow. Another problem with segment size selection occurred at sites that used gateways that would not accept large packets. Therefore, 4.3BSD chooses the maximum size segment according to the destination and the interface to be used. If the destination is local (on a directly-connected logical network), the size chosen is a convenient size near the maximum supported by the network. However, if the destination is not local, nothing can be determined about the nature of the path. In this case, the segment size is chosen conservatively, the lower of the default maximum size (576 bytes) and that supported by the outgoing interface. This value is used both as the maximum segment size offered to the sender by the receive side, and as the maximum size segment that will be sent. Of course, the send size is also limited to be no more than the receiver has indicated it is willing to receive. This change alone significantly increased performance across the ARPANET. Previously, under some circumstances ARPANET traffic was reduced to a trickle due to fragmentation and packet loss.

3. Congestion Avoidance and Control

The use of larger buffers frequently improves performance on long-haul networks as well as on low-delay local nets. When the path is lightly loaded, the sending host is able to send more before stopping to wait for a window update, and may receive the window update in time to continue sending without pause. However, the use of larger buffers can cause problems when bulk-data transfers must traverse networks with slow links and gateways with limited buffering capacity. The source-quench

ICMP message was provided to allow gateways in such circumstances to cause source hosts to slow their rate of packet injection into the network. While 4.2BSD ignored such messages, the 4.3BSD TCP includes a mechanism for throttling back the sender when a source quench is received. This is done by creating an artificially small window (one which is 80% of the outstanding data at the time the quench is received, but no less than one segment). This artificial congestion window is slowly opened as acknowledgements are received. The result under most circumstances is a slow fluctuation around the buffering limit of the intermediate gateways, depending on the other traffic flowing at the same time.

Network congestion problems were also reduced by fixing the sender silly-window syndrome avoidance strategy in TCP. Silly-window syndrome results when a receiver sends window updates when it can only accept a small amount of data, and the sender proceeds to send data in small pieces. In 4.2BSD, the send policy compared the amount that could be sent (the lesser of the data in the queue and the receiver's window) was compared to the offered window. Thus, small packets could be sent if the receiver offered a silly window. Once this was fixed, there were problems with peers that never offered windows large enough for a maximum segment, or at least 512 bytes (e.g., the peer is a TAC or an IBM PC). Code was then added to maintain estimates of the peer's receive and send buffer sizes. The send policy will now send if the offered window is at least one-half of the receiver's buffer, as well as when the window is at least a full-sized segment. (When the window is large enough for all data that is queued, the data will also be sent.) The send buffer size estimate is not yet used, but is desired for a new delayed-acknowledgement scheme that has yet to be tested. The change to silly-window avoidance exposed problems with the persist code, which worked only when the window was exactly zero. It has been fixed to persist when the window is small, and if no update has been received when the persist timer expires, a segment is sent with as much data as allowed. Subsequent timeouts must retransmit rather than entering persist state again.

Another change in 4.3BSD that is designed to reduce congestion in long-haul networks is an implementation of the algorithm proposed by John Nagle [Nagle84] to reduce the number of outstanding small packets. The algorithm is very simple: when there is outstanding, unacknowledged data pending on a connection, new data are not sent unless they fill a maximum-sized segment. This allows bulk data transfers to proceed, but causes small-packet traffic such as remote login to bundle together data received during a single round-trip time. On high-bandwidth, low-delay networks such as a local Ethernet, this change seldom causes delay, but over slow links or across the Internet, the number of small packets can be reduced considerably. This algorithm does interact poorly with one type of usage, however, as demonstrated by the X window system. When small packets are sent in a stream, such as when doing rubber-banding to position a new window, and when no echo or other acknowledgement is being received from the other end of the connection, the round-trip delay becomes as large as the delayed-acknowledgement timer on the remote end. For such clients, a TCP option may be set with *setsockopt* to defeat this part of the send policy.

A final set of changes designed to improve network throughput concerns the retransmission policy. The retransmission timer is set according to the current round-trip time estimate. Unfortunately, the round-trip timing code in 4.2BSD had several bugs which caused retransmissions to begin much too early. These bugs in round trip timing have been corrected. Also, the retransmission code has been tuned, using a faster backoff after the first retransmission. On an initial connection request where there is no round-trip time estimate, a much more conservative policy is used. When a slow link intervenes between the sender and the destination, this policy avoids queuing large numbers of retransmitted connection requests before a reply can be received. It also avoids saturation when the destination host is down or nonexistent. During a connection, when the retransmission timer expires, only a single packet is sent. When only a single packet has been lost, this avoids resending data that was successfully received; when a host has gone down or become unreachable, it avoids sending multiple packets at each timeout. Once another acknowledgement is received, the transmission policy returns to normal.

4. Network Management

Network planning and management for local networks of workstations and larger machines has changed somewhat since the introduction of 4.2BSD. Networks have grown larger and more complicated, and a larger number of vendors are represented on the average research network. These changes

make the subject of network planning and management more important and more difficult.

4.1. Subnet addressing

One factor that has alleviated some problems at larger sites while (at least temporarily) aggravating others is the adoption of an Internet standard subnetting scheme [Mogul84]. Subnet support allows a collection of interconnected local networks to share a single network number, hiding the complexity of the local environment and routing from external hosts and gateways. A local network may include multiple physical networks such as Ethernets, ring nets and point-to-point links, but for the entire network to appear as a single entity externally. This facility is used (and required) by a number of large university and other networks that include multiple physical networks as well as connections with the DARPA Internet. This simplifies routing to the network from other Internet sites, and reduces the routing burden on the core gateways. On the other hand, relatively few vendors have implemented subnet support as of yet. Mixed-vendor configurations including machines without source code may make it difficult to adopt subnetting uniformly. The presence of subnets also confuses the recognition of broadcast addresses.

Subnets are implemented by dividing the host part of the local Internet address into subnet and host portions. For each network interface, a network mask is set along with the address. This mask determines which portion of the 32-bit address is the network number, including the subnet, and by default is set according to the network class (A, B, or C, with 8, 16, or 24 bits of network part, respectively). Within a subnetted network each subnet appears as a distinct network; externally, the entire network appears to be a single entity. Conversion to subnets involves selection of the width of the subnet field, assigning subnet numbers to local networks, and then assigning new host addresses. By convention, the subnet field is a contiguous part of the host field adjacent to the network field. Although the 4.3BSD kernel will allow any width field for subnets, the user-level software works best with an 8-bit field on Class A and B networks and 4 bits on Class C networks. To avoid confusion with broadcast addresses, it is recommended that the subnet numbers of all zeros and all ones be reserved.

4.2. Broadcast addresses

Another important change in IP addressing in 4.3BSD is a change to the default IP broadcast address. A standard for Internet broadcasts has been issued which is different from the usage of 4.2BSD [Mogul85]. The default broadcast address is now the address with a host part of all ones (using the definition `INADDR_BROADCAST`), in conformance with the standard. In 4.2BSD, the broadcast address was the address with a host part of all zeros (`INADDR_ANY`). To facilitate the conversion process, and to help avoid breaking networks with forwarded broadcasts, 4.3BSD allows the broadcast address to be set for each interface. IP recognizes and accepts network broadcasts as well as subnet broadcasts when subnets are enabled. Such broadcasts normally originate from hosts that do not know about subnets. IP also accepts old-style (4.2BSD) broadcasts using a host part of all zeros, either as a network or subnet broadcast. An address of all ones is recognized as "broadcast on this network," and an address of all zeros is accepted as well. The latter two are sometimes used in broadcast information requests or network mask requests in the course of starting a diskless workstation.

4.3. Disaster avoidance

The two changes in Internet addressing described above make it very easy to produce degenerate situations in which broadcasts are either echoed or forwarded. In either case, degradation of network performance is dramatic. However, with some care in network and host configuration, it is possible to avoid these problems. Changes were made in 4.3BSD to avoid responding to broadcasts except when appropriate. For example, ICMP error packets are not returned in response to packets sent to the IP broadcast address, nor are such packets forwarded, but broadcast information requests are answered. In addition, 4.3BSD machines that have only a single hardware network interface never forward packets. Unless explicitly configured as gateways, they also do not send ICMP unreachable messages in response to packets that are not addressed to them.

Several examples will demonstrate the reasoning for these changes. The DEQNA Ethernet interface on the Microvax II is unusually sensitive to excessive collisions. At several sites, a group of

Microvax II's on a subnet have simultaneously dropped from the net until a software timeout causes resets. In each case, it was discovered that two groups of machines on the network disagreed about the broadcast address. In one case, each Microvax that received a network broadcast on a subnet simultaneously tried to forward the packet to the local gateway, causing a series of collisions. In other incidents, each 4.2BSD host on the network tried to forward such a packet to a (nonexistent) host on the same subnet; none had ARP entries for that host, and therefore all sent broadcast ARP requests for that address immediately after each network broadcast. Disabling IP forwarding improved matters somewhat, but each host sent ICMP unreachable messages to the originating host rather than attempting to forward the packets. The conclusions are clear: non-gateway hosts should take care never to forward broadcasts, and should not send error packets in response to broadcasts. All host IP implementations should support subnet addressing, and must be liberal in their recognition of attempts to broadcast.

4.4. Network layout

The layout of a collection of local networks and gateways may be a difficult problem. Initially, most organizations start out with a single Ethernet cable and some number of hosts. With the addition of large numbers of workstations, many of them diskless, additional network cables to extend the coverage of a network, and interconnections with outside networks with point-to-point links, the situation becomes much more complicated. The ability of 4.2BSD to forward IP packets has resulted in various strange network configurations, as VAXes with multiple network interfaces tie together groups of local nets. At one time, there were hosts within a single department at Berkeley that were 6 hops from each other, and routing loops existed due to multiple interconnections. The problems caused by random configuration can be minor, wasting resources with additional hops during normal operation or during routing transitions, or the failures may be catastrophic. For large or complicated configurations that have connections to the outside world, it is most reasonable to use subnets to restrict the details to local gateways. It is wise to place workstations which make heavy use of the network on separate cables from minicomputers and other hosts. One subnet should be used as a spine, to which other subnets are attached with a small number of dedicated gateways. The topology should never be more complicated than a tree structure.

4.5. Routing and use of 4.3BSD hosts as gateways

Several changes have been made in 4.3BSD in the area of gateway support (or packet forwarding, if one prefers). A new configuration option, GATEWAY, is used when configuring a machine to be used as a gateway. This option increases the size of the routing hash tables in the kernel. Unless configured with that option, hosts with only a single non-loopback interface never attempt to forward packets or to respond with ICMP error messages to misdirected packets. This change reduces the problems that may occur when different hosts on a network disagree as to the network number or broadcast address. Another change is that 4.3BSD machines that forward packets back through the same interface on which they arrived will send ICMP redirects to the source host if it is on the same network. This improves the interaction of 4.3BSD gateways with hosts that configure their routes via default gateways and redirects. The generation of redirects may be disabled with the configuration option IPSENDREDIRECTS=0 in environments where it may cause difficulties.

Several styles of routing are possible with a 4.3BSD host. The simplest, which works for singly-homed hosts in simple environments, uses a default gateway installed at boot time with a *route* command. Routes to networks or hosts that should use other gateways will be installed dynamically as the result of redirects received from the default gateway.

For more complicated environments, local area routing within a group of interconnected Ethernets and other such networks is probably best handled by the user-level routing daemon, *routed*. Considerable effort was expended to increase *routed*'s reliability, and the 4.3BSD version handles subnet routing along with routes to normal networks and hosts. Subnet routes are not sent to routers on a different network.

Gateways between the ARPANET or MILNET and one or more local networks require an additional routing protocol, the Exterior Gateway Protocol (EGP), to inform the core gateways of their presence and to acquire routing information from the core. An EGP implementation for 4.2BSD was done

by Paul Kirton while visiting ISI, and any sites requiring such support that have not already obtained a copy should contact Joyce Reynolds (JKReynolds@usc-isif.arpa) for information. That implementation works with 4.3BSD without kernel modifications. The EGP code needs minor modifications, as packets from the ICMP raw socket now include the IP header like other raw sockets in 4.3BSD. Sites that use both *routed* and EGP must take care that any overlap in routing domains is handled gracefully. The only way to accomplish this at present is to place routes in the startup configuration files for each router that override any incorrect or suboptimal routes otherwise installed by that router. Rather than propagating routes derived from EGP to hosts and subnet gateways on local nets, *routed* can direct local machines to use the EGP gateway as a default route; this is enabled by a command-line option to *routed*.

4.6. Address Resolution Protocol and Trailer encapsulations

The Address Resolution Protocol, ARP [Plummer82], is used by 4.2BSD and 4.3BSD to map 32-bit Internet addresses into 48-bit Ethernet addresses. This works well with other hosts that implement ARP, which now includes most common IP vendors. 4.3BSD includes modifications to ARP from SUN Microsystems that add operations to examine and modify entries in the ARP address translation table, and to allow ARP translations to be "published." When future requests are received for Ethernet address translations, if the translation is in the table and is marked as published, they will be answered for that host. 4.2BSD also included an old algorithmic translation from IP addresses when host numbers were greater than 1024; that has now been removed.

As the use of trailer encapsulations was a compatibility problem, 4.2BSD provided a boot-time option to disable trailers on a per-interface basis. Unfortunately, this was much too coarse a granularity. 4.3BSD hosts now negotiate the use of trailer protocols on the 10 Mb/s Ethernet on a per-host basis using ARP. Hosts desiring to receive trailer encapsulations must now so indicate by the use of ARP. This allows trailers to be used between cooperating 4.3 machines, while using non-trailer encapsulations with other hosts. The negotiation need not be symmetrical: a VAX may request trailers, for example, and a workstation may note this and send trailer packets to the VAX without itself requesting trailers.

5. Conclusions

The IP/TCP networking implementation in 4.2BSD has received wide use over the last several years. It works well under the conditions for which it was optimized. Since its release, it has found use in widely differing circumstances, and this experience has allowed the implementation to be refined. The changes introduced in 4.3BSD correct most of the problems that were experienced with use over long-delay paths and with gateways unable to handle full-sized Ethernet packets. Other compatibility problems have been redressed. In addition, 4.3BSD supports the recent additions to the Internet architecture, including subnet addressing and the newly-specified broadcast address. As 4.2BSD hosts are upgraded to 4.3 and vendors of 4.2 derivatives update their software, the management of mixed-vendor networks will become simpler. In the meantime, careful management of local networks will be necessary to allow graceful growth and prevent resource-wasting loops.

6. Acknowledgments

This work includes ideas and fixes from numerous 4.2BSD sites. Thanks are extended to the many users at Berkeley and elsewhere who test these changes, willingly or not.

This work was sponsored by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871 monitored by the Naval Electronics Systems Command under contract No. N00039-84-C-0089. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Government.

7. References

- [Leffler83] Samuel J. Leffler, William N. Joy and Robert S. Fabry, "4.2BSD Networking Implementation Notes," Computer Systems Research Group, U.C. Berkeley, July 1983.
- [Leffler84a] Samuel J. Leffler and Michael J. Karels, "Trailer Encapsulations," RFC-893, Network Information Center, SRI International, April 1984.
- [Leffler84b] Sam Leffler, Mike Karels, and M. Kirk McKusick, "Measuring and Improving the Performance of 4.2BSD," *Proceedings of the Salt Lake City Usenix Conference*, pp 237-252, June 1984.
- [Mogul84] Jeffrey Mogul, "Broadcasting Internet Datagrams," RFC-919, Network Information Center, SRI International, October 1984.
- [Mogul85] J. Mogul and J. Postel, "Internet Standard Subnetting Procedure," RFC-950, Network Information Center, SRI International, August 1985.
- [Nagle84] John Nagle, "Congestion Control in IP/TCP Internetworks," RFC-896, Network Information Center, SRI International, January 1984.
- [Postel81a] J. Postel, ed., "Internet Protocol," RFC-791, Network Information Center, SRI International, September 1981.
- [Postel81b] J. Postel, ed., "Transmission Control Protocol," RFC-793, Network Information Center, SRI International, September 1981.
- [Plummer82] David C. Plummer, "An Ethernet Address Resolution Protocol," RFC-826, Network Information Center, SRI International, November 1982.
- [Walsh84] Robert Walsh and Robert Gurwitz, "Converting BBN TCP/IP to 4.2BSD," *Proceedings of the Salt Lake City Usenix Conference*, pp 52-61, June 1984.

A Real-time Electronic Conferencing System
based on Distributed UNIX

Tatsuo Suzuki
Hideo Taniguchi
Hisayasu Takada

NTT Electrical Communications Laboratories
1-2356 Take Yokosuka Japan

Net:seismo!kddlab!nttlab!nttdpe!suzuki

Abstract

This paper describes a real-time electronic conferencing service constructed on a distributed UNIX operating system. In this conferencing service, identical contents are displayed in windows on each participant's work station and updated simultaneously. Using shared windows, any services which run on standard UNIX are available without any modification.

Basic mechanisms to support the conferencing environment are studied and implemented in the distributed UNIX. It has been developed on the basis of UNIX system V by networking the file system and introducing network directories which utilize the broadcasting ability of LAN.

The conferencing service has already been constructed and evaluated in a real machine environment.

1. Introduction

In the office, people spend much time on group communications such as meeting and discussion. Computer support systems for group work of distributed users are important[1] to modern offices.

This paper describes a real-time electronic conferencing system designed and constructed using an extended UNIX* system. It works on a distributed processing system, connected

* UNIX is a trademark of AT&T Bell Laboratories

by LAN, and integrates various media such as text, figure, image and phone grade voice. This distributed UNIX has been developed on the basis of UNIX system V by networking the file system and adding broadcasting capabilities, as well as multi-windows management functions. Basic mechanisms of a real-time electronic conferencing service have been implemented in part of the distributed UNIX kernel. The system efficiency is also evaluated and discussed.

2. Functions of conferencing services

One of the core services in office automation (OA) is a real-time electronic conferencing service performed amongst work stations (WS) connected by high speed communication lines such as LAN[2,3]. In such a service, it is indispensable to display identical contents on each participating WS, to update them simultaneously whenever necessary, and also to share telephone communications for discussions.

Various functions such as editing for a WS must be shared, and thus such services are referred to as shared services. The input for such shared services ought to be entered from a single WS that has the access-right i.e. the right to speak at that moment. The outputs of shared services should be distributed to each participating WS. Access-right alternates among participating WS's with a certain priority when required. In case of telephone communication, all participants may speak at any time without regard to access-right.

During a real-time conferencing, some conferees may use their WS's locally without being notified by other conferees, using one of the multi-windows in their WS's. Furthermore, they may transfer data from a local window to a shared window at any time.

Fig. 1. shows a proposed system architecture for the real-time conferencing service, where users at WS1, WS2 and WS3 are attending the conference.

3. Basic functions

In this chapter, basic functions necessary for real-time electronic conferencing service are defined. An important point is that shared services are not specially designed, but standard services such as "ed", "vi", Japanese document processors, and graph generators, which run on UNIX, are incorporated in the conferencing service without any modification. On the other hand, the real-time conferencing service is designed so that it may provide environments for them.

3.1 Broadcasting

Real-time electronic conferencing service must output shared information without delay to all participating WS's. Therefore, file I/O's, including window I/O's as special files, are needed to utilize the broadcasting ability of LAN.

3.2 Access-right control

Input information for shared services is entered from each WS at a variety of locations. If simultaneous entries from plural WS's were allowed, this would disrupt the input stream and hamper participants' comprehension. Therefore, input must be restricted to a single WS at one time, and WS's which have input information are forced to wait until access-right is given to them.

3.3 Voice communication via PBX

Real-time conferencing system must support various communication media. Since the present day LAN is unsuited to real-time voice transfer, PBX is incorporated for voice conversation. In other words, PBX-LAN dichotomy architecture is exploited. (See Fig. 1.)

4. Functions of the distributed UNIX

Most basic functions for real-time conferencing service are implemented as basic functions of the distributed UNIX. Particularly, the group I/O concept is introduced, utilizing the broadcasting ability of LAN, which is inherent to LAN.

4.1 Global tree

All resources within the distributed system are managed in the form of a tree structure called a global tree in the distributed UNIX. An example of a global tree viewed from node-N0 is shown in Fig. 2. Resources within other nodes are accessed via special directories, called network directories, N1 or N2. Each network directory corresponds individually to the local root of each node, and is created by "mknod" system-call.

A global tree consists of all resources viewed from a node. Thus, required network directories should be created in the global tree before using resources in other nodes. The control mechanism for the body of a network directory is described in reference[6].

In Fig. 2., discrete lines are used for local resources. Node n0 is the conventional (local) root directory. Node g0

(group network directory) is explained in the following chapter.

4.2 System-call transferring

System-calls to a remote resource are transferred to the node where the resource exists. An "agent kernel process" is created in the remote node corresponding to each calling process. The agent process executes the system-call locally.

This mechanism, shown in Fig. 3., is implemented as a part of UNIX kernel. The sequence of creating/destroying an agent process and the access command protocol are described in reference[5].

A system-call is determined to be remote when it contains a network directory in parameters such as a "path name" or a file descriptor. Otherwise it is local. The system-call is transferred to the node where the resource exists.

4.3 Group network directory

Group network directories are introduced in order to utilize the broadcast ability of LAN. Each group network directory, which is represented by a physical group address of LAN, corresponds to a group of nodes.

All accesses via group network directories are broadcast-type, and system-calls are transferred to all nodes of the group and executed simultaneously. For example, as shown in Fig. 2, by specifying the path name `"/g0/fl/a"`, files `"fl/a"` on both nodes `n0` and `n1` are accessed together at the same time. Applying the network directory concept to simultaneous display on a group of nodes in real-time conferencing is very effective.

The access algorithm to a group network directory is the same as to a single network directory. An example command sequence of group remote access is shown in Fig. 4.

4.4 Access-right control

One of the problems using group network directories is how to treat input-type system-calls such as `"read"`. If the system-call `"read"` were to go to all members of the group and executed, read data would be returned from all members. It would be possible to attempt to use multiple entries, but for the original process that issued the `"read"` system-call, the following problems would occur.

- 1) a reply format differing from that of conventional `"read"`,

- 2) an undeterminable end point of the system-call unless the number of nodes in the group is known.

For these reasons, it is decided that only one specified node executes the system-call and returns its result. In order to assign the access-right node dynamically by software, a new system-call "setnod" is added.

As the access-right corresponds to "the right to speak" in the real-time conferencing service, it can be applied to control the real-time conference. "Setnod" system-call can be also used in case of writing on a single node of the group.

4.5 Process control

To extend process control system-calls such as "fork" and "exec", there are two cases, local and remote. Local "fork" and local "exec", which inherit the process environments containing remote accessing file descriptors, have less difficulty to be implemented. On the other hand, for remote process control, a new basic system-call "warp" is implemented, instead of extending "fork" and "exec" independently.

When a process issues the system-call "warp", it moves to another machine and continues its processing, while its environment such as file descriptors and the parent-child relation are preserved. It also works on a network directory and broadcasts the process itself to a group of nodes, though it does not inherit the environment in this case.

Note here that "warp" contrasts to file migration, by which the efficiency of resource access is improved by placing the process near the resources.

5. Structure of real-time conferencing service

The real-time conferencing service consists of the following four phases. Planning phase is performed by using other OA services before the conference begins. Setting-up phase sets up the environment for the conference, in which shared services are available using shared windows. In progress control phase, access-right is controlled. Completion phase terminates the conference and clears up the environment.

5.1 Planning phase

During the planning phase, the conference name and participants are selected, the schedule is adjusted, and invitation mails and reference documents are sent using other service functions of WS.

5.2 Setting-up phase

After confirming participants and selecting a chairperson, real-time electronic conferencing is set up to create the environment. The set-up program completed in one WS, e.g. the chairperson WS, is to perform:

1) group-path setting

- * To arrange for a physical group address of LAN.
- * To create group network directories in all participating WS's by issuing remote "mknod" system-call serially.
- * To move the "root directory" to the newly created group network directory by issuing "chroot" system-call.

Once this has been set up, all file I/O's (except the path names that begin with "/", i.e. the network root) are treated as a group I/O.

These environments are kept in the conference desktop manager, a kind of "shell" which manages all services and data in the form of icons. Any services, which are activated in the shared conference desktop, inherit all environments stated above.

2) access-right initialization

At the beginning of a conference, the chairperson has the access-right.

3) shared-cursor initialization

The shared cursor, displayed simultaneously in all participating WS's on the same position, is used in addition to the usual (local) cursors. Shared and local cursors coincide at the WS having the access-right.

4) setting up conference telephones

The audio part of the conference is initiated by dialing to the specially assigned prefix and the destination numbers, and then each participating telephone is connected to a conference bridge in PBX.

Shared services control process, shared cursor displaying process, access-right control process, etc. are activated after setting up all environmental elements; then the conference begins.

5.3 Progress control phase

Access-right assignment is changed, when required,

according to priorities, and the shared cursor is controlled so as to follow the local cursor of the WS to which the new access-right is assigned.

During the conference, each participant can use their local services independently by activating services on a local desktop. Windows made by local services are not broadcast and can not be seen by other participants. Furthermore, participants can transfer the data from a local window to a shared window and broadcast them by using an inter-window transferring function available in the local multi-window system[7].

5.4 Completion phase

All kinds of processes are terminated and the group path is released by removing the group network directory, all by command of the chairperson, to finish the conference. If necessary, the minutes of the conference are circulated.

6. Evaluation

The conference service has already been constructed and evaluated in a real machine environment.

The overhead of group read/write access time, compared to local access time, is about 20~30 msec. It depends on the number of packets in which commands/data are carried through LAN by CSMA/CD method. But in most cases, all data for an access are included in one packet.

For some OA services such as OA document system, response time is measured in the case of shared services of an electronic conference. The average time ratio in comparison to local execution is about 1.2, or 20% overhead.

7. Conclusion

In the distributed UNIX, the file system kernel is extended to the network without changing interfaces and with group I/O introduced. Most basic functions needed for real-time electronic conferencing service are implemented as kernel functions. These modifications are easy to implement.

The system has already been constructed and evaluated in a real machine environment.

Real-time control of the conference is achieved, because the overhead of group remote access (such as read/write) is about 20msec/access. It has been shown that usual OA services work well as shared services in this electronic

conference system.

8. Acknowledgements

The authors would like to thank Yoichi Kishimoto for his support in evaluating the system. They are also grateful to their managers, Dr. Naohiko Kamae and Keiichi Kawada, for their editorial suggestions and encouragement.

References

- [1] Sarin, S. and Greif, I., "Computer-Based Real-Time Conferencing Systems", Computer, vol. 18, no. 10, pp. 33-45, 1985.
- [2] Sakamoto, Y. and Suzuki, T., "An Integrated Office Communication Service", National Conference of the Institute of Electronics and Communication Engineers of Japan, 1982 (In Japanese).
- [3] Suzuki, T., "Real-time Bi-directional Communications by window sharing", 28th National Conference of the Information Processing Society of Japan (IPSJ), 1983 (In Japanese).
- [4] Rowe, L. A. and Birman, K. P., "A Local Network Based on the UNIX Operating System", IEEE Trans. Software Eng., SE-8(2), pp. 137-146, 1982.
- [5] Suzuki, T. and Taniguchi, H., "Remote Access functions of OS for LAN - an experiment on UNIX -", 28th National Conference of IPSJ, 1984 (In Japanese).
- [6] Taniguchi, H., Suzuki, T. and Zeze, M., "A Distributed Operating System Based on Networking the File Management Mechanism", IPSJ Trans., vol. 27, no. 1, pp. 56-63, 1986 (In Japanese).
- [7] Yokoyama, S. and Yamada, S., "The Facility of Data Transfer between Windows on the Multi-Window System", 30th National Conference of IPSJ, 1984 (In Japanese).

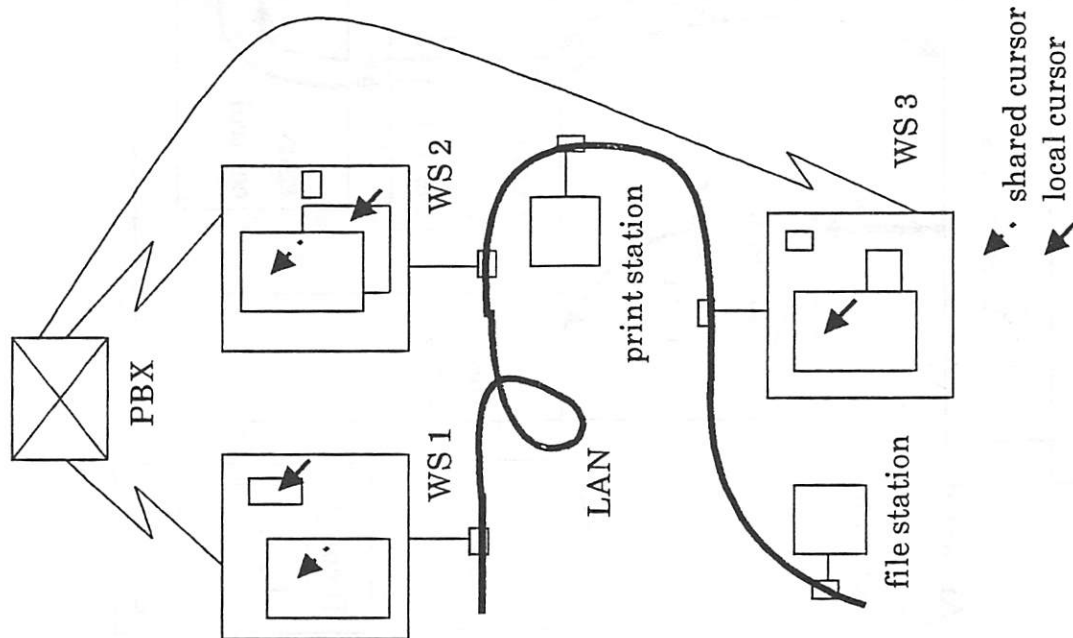


Fig. 1 Real-time conferencing

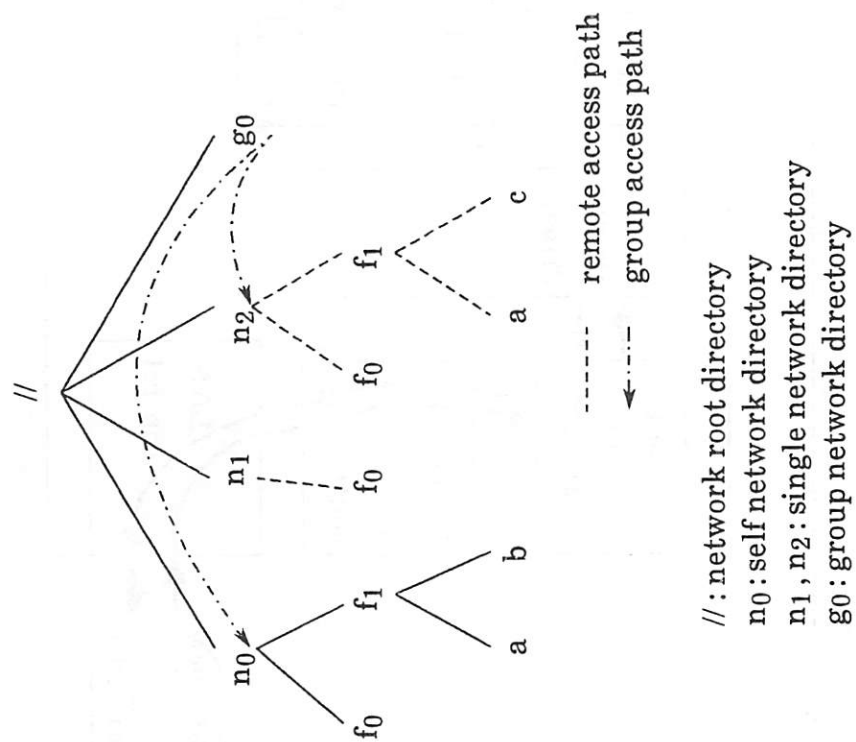


Fig. 2 Example of global tree (viewed from node n_0)

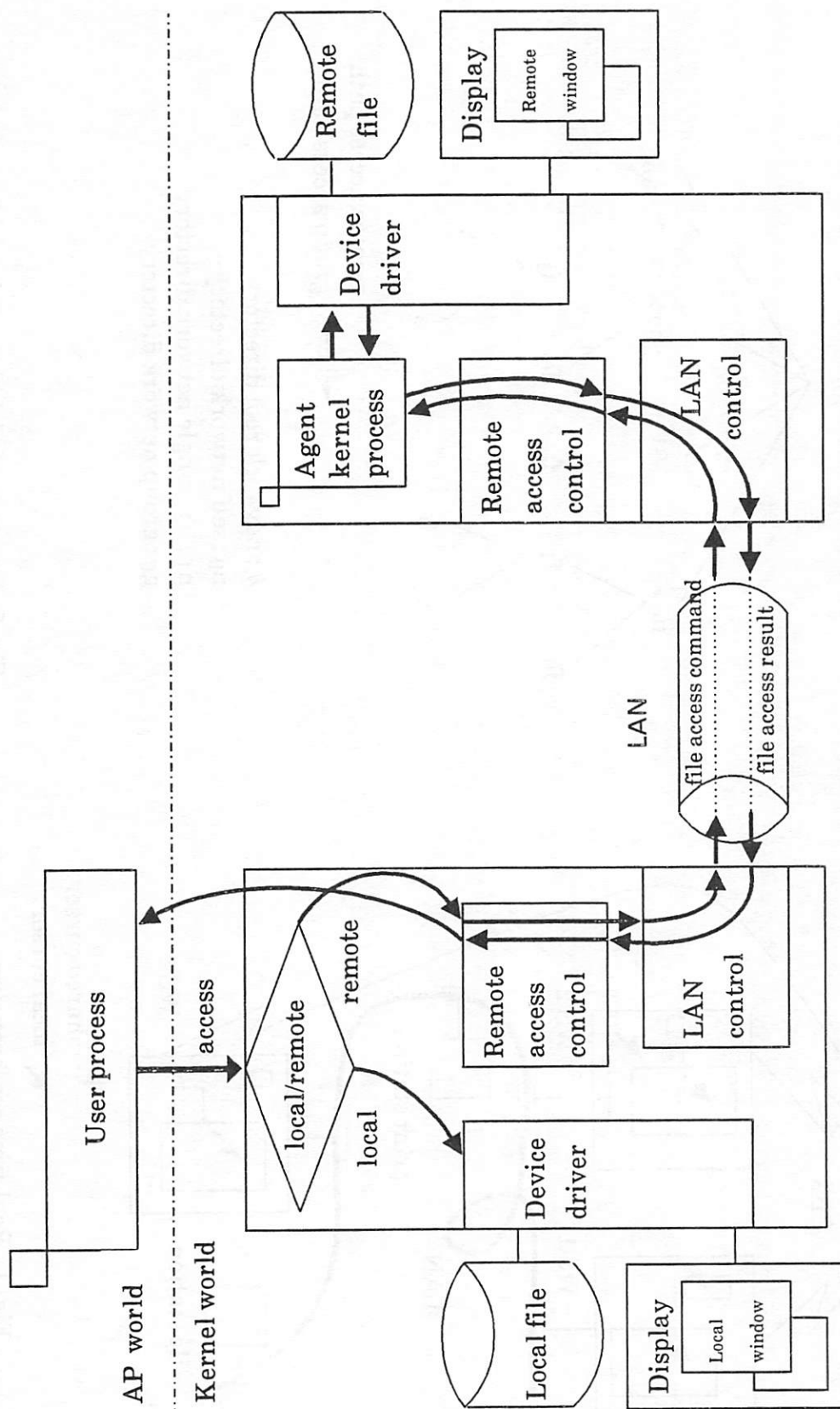


Fig. 3 Overview of remote access mechanism

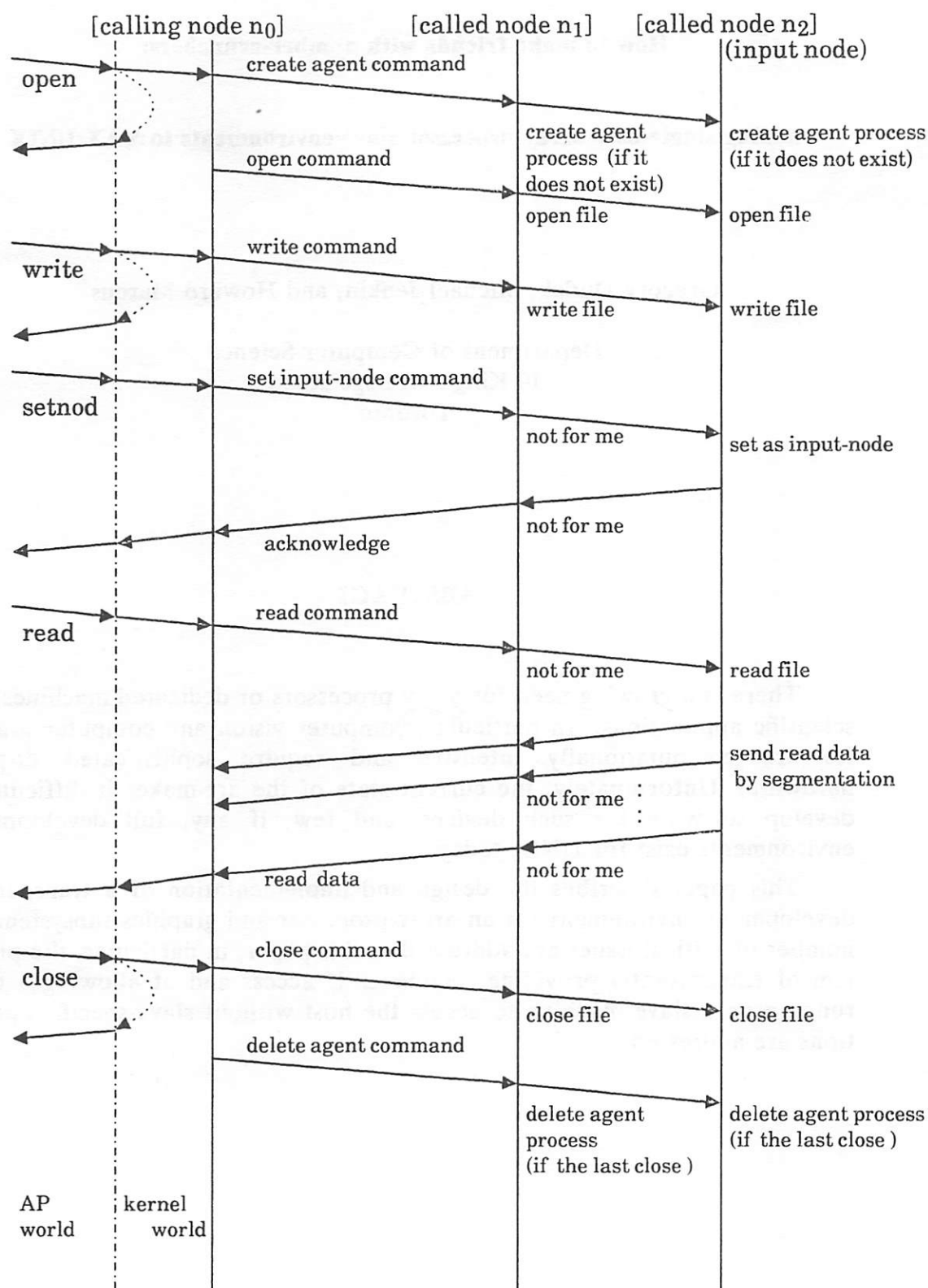


Fig. 4 Example sequence of group remote access

How to make friends with number-crunchers:

adding single-user array-processor slave environments to VAX UNIX

Gregory Dudek, Michael Jenkin, and Howard Marcus

**Department of Computer Science
10 King's College Road
Toronto**

ABSTRACT

There is a growing need for array processors or dedicated machines for scientific applications. In particular, computer vision and computer graphics are computationally intensive and require sophisticated display hardware. Unfortunately, the current state of the art makes it difficult to develop software for such devices, and few, if any, full development environments exist for UNIX today.

This paper describes the design and implementation of a transparent development environment for an array-processor and graphics subsystem. A number of critical issues are addressed in this paper; in particular, the problem of transparently providing remote CPU access and of allowing a task running on a slave machine to access the host without slave-specific operations are addressed.

1. Introduction

Array-processors are often touted as the solution to the problem to numerical calculation intensive tasks. The task of producing code to execute on an array processor can be very time consuming. Array-processors often have very non-standard instruction sets, and few, if any, support the execution of complete high-level language programs. Most vendors of array-processors (including the MSP-3000), provide packages which allow programs executing in the VAX environment to perform single vector operations on data which must be explicitly down-loaded and up-loaded to and from the remote array processor. Since conventional debugging tools are often inapplicable to such code segments, it can be particularly difficult to develop and test program which take advantage of the computational power of an array processor.

In a radical departure from such systems, we have developed a package that allows code to be developed in 'C' under UNIX. This code can be tested, and debugged using all of the standard UNIX tools. Once the code has been debugged, it can be re-compiled and executed on the array-processor without change to the source code. Code need only be recompiled and linked to execute remotely.

In the process of developing this system, a number of interesting problems were addressed. In particular, that of developing a high level language ('C') that was 100% compatible with UNIX 'C', allowing for transparent execution (by the user) of code on the array-processor, and the task of allowing executing code to access user and system level functions that are available under UNIX even though the code is executing remotely on the array processor.

2. Hardware Description

The software has been developed for the MSP-3000, a high-speed user-programmable full floating point array processor which is plug compatible with Digital Equipment Corporation's PDP-11, and VAX series computers. The machine is capable of performing a 1024-point complex FFT in under fourteen milliseconds. The MSP-3000 can be optionally fitted with a raster display processor, capable of displaying up to 512 lines of 512 pixels of data, with 12 bits of colour per pixel.

At the University of Toronto there are 3 MSP-3000's, each of which is equipped with 2 meg of memory, a raster display processor, and a keyboard complete with trackball. The devices are used by a number of different users, both as 'dumb' display devices, slaved to a VAX task, as well as the primary processor for major computer vision related tasks, such as floating point image convolutions and segmentation algorithms.

The MSP-3000's instruction set supports a limited number of general purpose instructions, based around two fundamental data types, a 32 bit floating point value (compatible with the PDP-11 floating point format), and a 32 bit integer value. Some support is provided by the instruction set for byte, short word, and bit accesses. In addition, a table of entry points is provided in EPROM to the array processor instruction set. These entries provide support for both real and complex vector operations.

3. System Structure

The software system that we have designed and implemented allows a user to write code that is independent of the special hardware available on the array processor. To test, and debug this code on the VAX, and then to re-compile and execute the code on the array processor. In order to allow the user to write code that takes advantage of the processing speed of the vector operations supported by the MSP-3000's, libraries have been written to directly access these operations on the MSP-3000, and to emulate them in a non-vector manner on the VAX. Thus code can be written in a 'vectorized' form on the VAX, and then executed on the MSP-3000's.

In order to allow the user to perform these actions, a number of sophisticated software tools were written. Not only was it necessary to generate executable code for the MSP-3000's from full 'C', it was also necessary to allow a user to execute code on the array processors without having to re-write all system and library calls. In order to simplify the problem of actually down-loading and starting the execution of processes on the array-processor, a custom shell was written (based on the standard *csh*) which recognizes executable modules for the array processor, and which automatically down-loads and executes them on the array processor. There is support for both system calls, and the standard 'C' libraries. In addition, there is support for the specialized hardware that is available on the array processors. Note that these specialized operations are emulated during program testing and debugging on the VAX UNIX machine.

3.1. Software Development Tools

In order to produce native code for the array processor, a number of tools have been developed.

o Cross-compiler

In order for the array processor's 'C' compiler to be 100% compatible with the portable 'C' compiler available under UNIX, a cross assembler was written which translates individual VAX assembler instructions to MSP-3000 assembler code. A number of the more complex instructions available in the VAX instruction set (*ediv*, for example), are actually translated into subroutine calls. This cross assembler emulates the instruction set available on the VAX machine.

Due to the limited instruction set on the MSP-3000 (as compared with the instruction set on the VAX), a number of operations may take significantly longer to execute on the MSP-3000 than on the VAX. For example, the standard strcpy function will take considerably longer to execute, as bytes must be extracted from 32 bit words, and then re-packed into the destination bytes. In order to improve throughput, a number of the standard UNIX libraries were hand-coded in the MSP-3000's native instruction set.

The cross-compiler mimics the VAX register and stack structures, as well as allowing for argument and environment strings to be passed to the executing routines. The cross-compiler requires a library of MSP-3000 routines to mimic certain difficult operations, to initialize the MSP-3000 hardware, and to initialize the emulated VAX stack and free memory.

o MSP-3000 Assembler

Once MSP-3000 assembler code has been produced, it must be assembled. The MSP-3000 assembler produces object modules with the identical format used under the host UNIX environment. This allows the use of the host UNIX loader, and object file utilities for the manipulation of the array-processor object files. In particular, it allows for the use of the standard VAX UNIX loader to link object modules together and to resolve external references. A small modification is made in the header of the object file so that the resulting file can be distinguished from normal VAX UNIX executable files. In order to prevent a user from accidentally executing a MSP-3000 executable file on the UNIX machine, a UNIX compatible entry point is added to the object file which prints out the message CDA CODE and then exits.

o UNIX Device driver

Recognizing the inefficiency of performing a system call and context switch for each access of the MSP-3000, most of the communication with the device is performed in user space. The UNIX device driver simply allocates and returns the address of the register block of the MSP-3000 allowing direct access of the registers from a user's program. Furthermore a common block is allocated by the driver for use as dual port memory, and the address of this area is also returned to the user for fast transmission of data between the host and the array processor.

o MSP-3000 csh

In order to provide truly transparent access to the array processor for users, a modified version of the C-shell command processor has been developed. This shell allows users to execute an array processor program as if it were normal UNIX code. This is accomplished by modifying the header block of array processor programs so that they can be identified. A modification that allowed them to be identified while retaining

compatibility with normal system utilities proved to be rather elusive. Simple approaches such as using a distinguished "magic number" are undesirable since many utilities refuse to handle files with non-standard magic numbers. This would have been highly undesirable since it would have made it impossible to use such standard software tools such as those for printing symbolic maps or doing string table optimization. The approach that was finally chosen consisted of distinguishing array processor programs by giving them a combination of an unusual magic number and a very unusual starting address.

Users of the array processor define an environment variable in the shell which indicates the path to the array processor interface program. When subsequent commands are executed by this shell, the header block of each file to be run are inspected just prior to their execution. If the header indicates that the program is array processor code, then the MSP-3000 interface is executed rather than the user code, and the user code is down-loaded for remote execution. The choice of which array processor is to be used can be specified explicitly by the user via another environment variable or, failing this, a free array processor is selected automatically by the interface.

In the case of users who have not set the appropriate environment variable to specify automatic remote execution, MSP-3000 programs are linked so that if they are executed on the VAX via an "exec" call, the starting address of each module points to a small VAX code routine which simply prints an advisory message and exits. The array processor interface can then be invoked explicitly.

o MSP-3000 symbolic debugger

The MSP-3000 symbolic debugger is a multi-purpose package which fulfills four major functions. It serves as a loader for programs which are about to be executed on the MSP-3000, it functions as a hardware-level debugger and diagnostic package, it plays the part of a software-level debugger something like UNIX's "dbx", and it serves as an interface between programs running on the MSP-3000 and the host UNIX system.

The low-level hardware and software features of the debugger include the ability to examine and modify all the registers, control signals, and memory locations of the array processor. This extensive control proves useful both for developing applications as well as doing hardware maintenance and system software development.

Higher level features supported by the debugger parallel those of the UNIX debugger "dbx". Supported functions include the insertion of break-points, the tracing of assembler instructions as they are executed, the tracing of source code lines (across multiple files), and the ability to execute selected portions of code.

The program loader support provided by the debugger allows users to down-load pre-compiled UNIX-format object modules to the array processor. It also allows loading of the symbol table alone so that existing array processor memory can be examined symbolically, much like a UNIX core file.

The UNIX system interface can also be initiated, examined, and traced using the debugger. This includes the tracing of all system calls, or the listing of arguments to each call. The details of the interface are discussed below.

3.2. Library Support

There are two major sets of software libraries that have been written for the array processor. The VAX set, is a collection of libraries for code that is to be executed on the VAX under UNIX. Two major groups of libraries have been written. The first allows device independent raster graphics to be displayed using the MSP-3000's display processor. This raster graphics library (rastersoft), also supports several other display processors available at the University of Toronto. The second group allows a user to mimic the array processor instructions that are available on the MSP-3000. This library supports such functions as vector add, vector subtract, etc.

The second set of software libraries support operations that take place on the MSP-3000. These libraries provide support for the standard 'C' libraries, including the string, math, and standard I/O libraries. Interestingly enough, due to the use of a cross assembler as the underlying tool for production of code on the MSP-3000, almost all VAX libraries could be simply recompiled for the MSP-3000. In practice, however, a rewriting of the libraries was performed in order to produce smaller and faster code. The math library, for example, was re-written to use the underlying vector operations to determine the required results; ie, the sqrt routine constructs a small vector with an element with the passed argument, and then calls the vector library entry for sqrt to obtain the required result.

These libraries are a crucial aspect of the ability to maintain the interchangability of the local and remote execution environments.

4. Interfacing with a remote UNIX

As noted above, the full range of UNIX system calls are supported for jobs running on the array processor. This is accomplished by a processes on the UNIX system which acts as an agent on the behalf of the MSP-3000 program. For each process running on an MSP-3000 processor there is one agent process dedicated to carrying out its requests. The communication between the MSP-3000 and the UNIX agent process is invisible to user processes. Programs execute system calls as if they were running on conventional UNIX systems; the inter-machine support requires no special

precautions or parameters from the user.

Most system calls are implemented on the MSP-3000 by preparing a buffer containing the call's parameters and within the MSP-3000 system support code and then signalling the VAX. The agent process on the VAX reads the buffer, performs the appropriate system call on the VAX, and returns the necessary values. In a sense, MSP-3000 programs are thus composed of two parts: a main code section that runs on the array processor, and a small agent process that maintains the program's UNIX context. This structure makes maintenance of a normal UNIX interface quite straightforward. Since all the contextual information normally associated with UNIX processes (working directory, pipes, etc.) is maintained by the agent process, the operating environment for user processes on the array processor is no different from that of normal UNIX programs.

Alternate strategies for system support for the array processor would have been to have the driver code within the system carry out the requests directly, or to have a single agent process for all processes executing on *any* remote machine. In the former case, there would be some advantage in terms of simplicity since the kernel-user boundary would not have to be crossed on the host system for every system call (saving context switching and validation). On the other hand, the overall complexity of the design would be much greater due to the special-purpose features that would have to be incorporated in the driver, the ease of development, especially incremental development, would be drastically reduced since changes would require kernel rebuilding, and the overall reliability of the resulting system would be far more questionable since array-processor requests would have direct access to the internals of the kernel.

Using a single agent process for all remote machine would have the advantage of reducing the number of processes on the host UNIX machine, but would entail greater complexity than the approach used since each remote process' context would have to be maintained explicitly by the user code.

The disadvantage of having one agent process for each array processor job is not particularly severe. Since all the processes share the same code segment, only one copy need be in memory on the host. In addition, most agent processes spend the majority of their time waiting for a request from the MSP-3000 and can thus have their data spaces swapped out. For the same reason, they impose little CPU load on the host machine.

5. Comparative System Performance

Tasks built to run on the MSP-3000 which make heavy use of resources available on the remote UNIX system will run considerably slower than the same task executing on the VAX. Certain data manipulation tasks will, in

particular memory accesses that refer to non long word or floating point data items, will take longer to execute on the MSP-3000 than the similar operation would on the VAX. Despite this fact, the lack of other users on the MSP-3000 will allow for a throughput that is generally independent of the load on the VAX, and we have found that the response time on the MSP-3000 is in general faster than that on the VAX.

Benchmark programs have shown that programs dependent on simple numerical calculations execute substantially more quickly on the MSP-3000 than on the VAX. Programs that can be rewritten using the array operations will result in dramatic decreases in processing time (and UNIX charges). Unfortunately, tasks which are heavily dependent on UNIX system calls (file access, process status, etc.), suffer a penalty in terms of performance. Preliminary estimates suggest that the overhead for system calls from the array processor is four times that incurred by jobs executing on the VAX.

In considering system performance, it is important to consider the effect of adding array processors to the other users of the system. Unless the task running on the MSP-3000 is a heavy consumer of UNIX resources (file I/O or other system calls), the MSP-3000 will run independently of any and all users on the VAX. Indeed, the MSP-3000 will even run when the host machine is down. The removal of computationally intensive tasks from the job mix of the VAX machine will allow tasks to run more quickly on the VAX, while allowing the task running on the MSP-3000 to execute independently, and to take advantage of the specialized vector and display hardware without interfering with other tasks running in the VAX environment.

6. Conclusions

The paper addresses some of the problems and issues involved in developing slave systems serving under UNIX host machines. The design proposed is simple to implement, and can be designed incrementally, allowing for varying degrees of autonomy on the part of the slave processor. The use of UNIX format object files, the UNIX assembler, linking-loader, and other development tools allowed for a quick and convenient implementation of much of the MSP-3000 software. By constructing a device driver that performs most of its functions in user space, the testing and debugging of "system" software can be performed without inconveniencing regular users of the host machine, and hence greatly accelerates the development cycle.

The implementation of this system has allowed tasks to be executed that perform computational tasks that would have swamped the host machine, much to the satisfaction of not only the owner of the task, but also to the delight of other users of the host machine.

7. References

Dudek, Gregory, and Hamacher, Carl, and Holt, Richard C., *The Design of a Small Distributed System: A Practical Evaluation*, Canadian Information Processing Society congress, June 1985, Montreal, Canada.

Kanji UNIX: Yunikkusu wa Nihongo o Hanasemasu (UNIX Speaks Japanese)

Robert S. Jung
Manager UNIX Ports
UniSoft Systems
ucbvax!unisoft!bobj

Joseph T. Kalash
Manager Special Projects
UniSoft Systems
ucbvax!unisoft!kalash

UniSoft Systems
739 Allston Way
Berkeley, CA 94710

ABSTRACT

UNIX [RIC78] was born and raised in the United States, but in the process of growing up UNIX has had to become much more worldly. However, until recently there have been few UNIX systems that could speak more than one language (English). In this paper, we examine a new system that speaks not only English (ASCII codes), but Japanese (Japanese Information Standard codes) as well. A description of the system's Japanese language capabilities is presented along with discussion of the design goals and implementation details.

1. Introduction

A UNIX system for the Japanese language was developed by AT&T UNIX Pacific and released in January of 1986. This product is officially known as the UNIX System V Japanese Application Environment (JAE) [ATT86]. In this paper, it is also referred to as Kanji UNIX.

The experience and knowledge gained from developing Kanji UNIX can be beneficial to UNIX system implementations for other native languages. The Kanji UNIX design took special effort to implement basic functionality that would be applicable to UNIX systems supporting languages other than Japanese.

UniSoft Systems was contracted to assist in the design and specifications as well as to do the development of the UNIX kernel enhancements for this product. Nippon UniSoft Corporation (NUC), which is a UniSoft System's Japanese joint venture, was contracted to work on the dictionary daemon, dictionary maintenance programs, and the Japanese version of the vi editor. Four other Japanese computer companies were contracted to work on other aspects of this project. The majority of the development was done in AT&T UNIX Pacific's office in Tokyo.

This paper focuses primarily upon the kernel level design and development issues encountered during the Kanji UNIX project, although some of the issues and resolutions of user level programs and interfaces will be covered.

2. Functionality of Kanji UNIX

2.1 What is Japanese?

The Japanese language has two alphabetic systems (katakana and hiragana) and one ideographic system (kanji). There are a total of 96 characters in the alphabetic systems (48 in both katakana and hiragana), and about 4000 kanji characters.

Hiragana is used for writing words of Japanese origin, and for extending the meaning of the Kanji characters. Katakana is used to write foreign words imported into the Japanese language.

As katakana and hiragana are both phonetic alphabets, it is possible to use the English alphabet to write equivalent sounds. Using the English alphabet to write Japanese sounds is called romaji.

The Japanese Information Standard (JIS) code set includes codes for hiragana, kanji, katakana and a half-width version of katakana called kana.

2.2 Input and Translation

With a standard Japanese terminal, there are two methods to input kana:

- kana via direct input from special keys on Japanese language terminals
- kana via romaji which uses the normal ascii characters.

Kanji UNIX provides several ways to manipulate this data:

- katakana via kana
- katakana via romaji
- hiragana via kana
- hiragana via romaji
- kanji via kana
- kanji via romaji
- kanji via kuten (specifying the 16-bit code directly).

In addition, Kanji UNIX also supports input of EUC directly from a personal computer with Japanese language capability.

2.3 In-Place Translations

User definable metacharacters are used to enable and disable the various translation methods. Except for kanji via romaji, and kanji via kana, the above translations are performed in-place under Kanji UNIX (See Figure 1.), instead of on a translation line at the bottom of the terminal screen as is done in other Japanese language computer system implementations.

Except for kanji via romaji, the translations are done either upon entering new-lines, or upon request. The user can translate a kana string into hiragana by typing a ^G. All of the other translations have a similar control character to handle the translation (See Figure 2.). All of the control characters are settable with either ioctl calls, or the stty command.

2.3.1 Kanji Translations

For kanji via romaji/kana, in-place translations are not possible. Although each kanji character has only one kana representation, most words written with kana characters can represent more than one kanji. This is because many kanji characters sound the same, but are written differently.

In order to perform the translation, it is necessary to have the user make a selection from the possible kanji characters. A dictionary process looks up the possible kanji characters and displays the selection on a dedicated line of the terminal (called the “guide” line). The following is how a user interacts with Kanji UNIX to enter kanji data (See Figure 3.):

1. The user enters a kana sequence.
2. The user requests the system to translate a kana sequence into kanji.
3. The kernel displays upon the status line the kanji's that are phonetically equivalent to the kana sequence.
4. The user types some response (either a selection of one of the kanji's displayed or a request for more choices).
5. If the user asks for more choices, the system goes back to step 3.
6. If the users selects one of the kanji's, the system replaces the kana sequence with the selected kanji.

2.4 Standard UNIX Functionality

Kanji UNIX provides all the functionality of standard UNIX System V. Its added functionality is provided as add-on kernel modules to System V and a set of utilities and commands. Once these modules are loaded into a kernel, then basic UNIX will appear the same to the casual user, while a user of the Japanese capabilities will have extra capabilities that will not interfere with the standard system.

2.5 Japanese Terminal Support

Kanji UNIX was designed to run on as many terminals (both standard ASCII and Japanese) as possible. Doing this required setting up various terminal specific strings in the kernel that can be down loaded for individual terminals. For example:

- Enter Kanji display mode.
- Exit Kanji display mode.
- Initialize guide line (this is the line for choosing kanji's).
- Goto the guide line.
- Leave the guide line.

All of these parameters were added into the **terminfo** database. All terminals with Japanese capability are stored with the extension '-j'. So an Itochu Electronics cit600 terminal, is known as a cit600-j in **terminfo**. This allows for both a standard terminal description for a terminal, if the Japanese capability is not wanted, and a description which allows the capability.

Two types of personal computers, three types of terminals, and one type of printer are supported as Japanese language i/o devices by the standard **terminfo** database files provided by the first release of Kanji UNIX [ATT86].

3. Design and Implementation of Kanji UNIX

There were four basic goals in the overall design of Kanji UNIX.

- The final system should be adaptable to other native languages that require two-byte code characters.
- The Japanese capability should not interfere with standard UNIX.
- Programs written to adhere with the System V Interface Definition should work under Kanji UNIX.
- The Kanji UNIX system must be compatible with existing and future releases of UNIX.

The design issues for the kernel enhancements included: input methods for Japanese alphabetic and ideographic written languages, supporting Japanese language terminals, 8-bit and 16-bit data paths through the kernel, data representation, dictionary look-up ability, System V consistency, and configuration.

3.1 Data Representation of Japanese

Hiragana and kanji are both represented by two-byte codes (as opposed to ASCII's one-byte codes). The external data representation (input) recognized by Kanji UNIX are the standards JIS-C6226 and JIS-C6220 (referred to as JIS in this paper).

Internally, Kanji UNIX uses an Extended UNIX Code (EUC) to represent and manipulate Japanese data. EUC is two bytes of data, with the upper bit on in both bytes, to differentiate it from ASCII.

Katakana exists as both one-byte codes (henceforth referred to as simply kana) and two-byte codes (henceforth referred to as katakana). However, only the kernel knows about one-byte kana characters. All kana data sent to programs, or stored on disk is in EUC format. This is done to tell apart two one-byte kana's from one two-byte EUC.

The JIS character set includes codes for all 8-bit kana, 16-bit katakana, 16-bit hiragana, and 16-bit kanji.

3.2 Kanji Line Discipline

The cleanest approach that met the design goals was to design a new line discipline. A separate line discipline allows users who do not wish to use the Kanji UNIX features to use the default line discipline provided by System V. Those who need the Kanji capabilities can access them through the standard UNIX feature of the line discipline switch.

The Kanji line discipline has all the features of the default line discipline and can be seen as an extended version of it. This means that the Kanji line discipline supports features such as escaping metacharacters and handling breaks and interrupts.

The added functionalities of the Kanji line discipline are to

- perform translation of 8-bit kana to 16-bit katakana,
- perform translation of 8-bit kana to 16-bit hiragana,
- perform translation of kuten to 16-bit EUC
- perform translation of romaji to 8-bit kana,
- provide a message passing protocol for user level dictionary daemon processes

The first three types of translations are one-to-one mappings and are straightforward to implement. However the translation of romaji to 8-bit kana is a more complex operation. There are a small number of possible romaji's and each is from one to four ASCII characters in length. When data is input with romaji translation mode enabled, the data is stored into a special romaji buffer and one of three actions will occur:

1. the buffer contains a potential legitimate romaji sequence (but incomplete), the data will be echoed as ASCII, and left in the buffer.
2. the buffer contains a complete romaji sequence, the buffer will then be cleared and the appropriate kana is then placed in the input queue and the displayed romaji sequence is then replaced by the kana.

3. the buffer contains a sequence that cannot be a legitimate kana, and then the portion of the buffer which cannot be part of a romaji sequence is moved to the input queue. The input character is then echoed as ASCII.

There are three restrictions upon the kana to kanji translation system that makes it impractical to do the kanji translation in the kernel.

- The dictionary which stores the information for translation is approximately one-half megabyte.
- The requirement that different translation schemes be easily implemented.
- The ability to switch dictionaries easily.

The implementation decided upon was to use a user level “daemon” to handle translations from kana to kanji. This daemon initializes with an ioctl call (TCJDAEMON) to tell the kernel it is there. The algorithm used to interact with the user for translation is as follows:

1. The user requests the system to translate a kana string into kanji.
2. The kernel “writes” the kana string to the daemon.
3. The daemon looks up the kana string in the dictionary.
4. The daemon sends a terminal line worth of kanji’s to the kernel.
5. The kernel prints out the kanji’s, handling entering/exiting the status line.
6. The user types some response, that response is sent back to the daemon.
7. If the user asks for more choices, the daemon goes back to step 4.
8. If the users selects one of the kanji’s, the daemon writes the data to the kernel, then the kernel replaces the kana string with the kanji.

3.3 Implementation Considerations

Many of the difficulties in implementing the Kanji line discipline arose because there were several types of data and the line discipline is translating data into different types within its input queues. The data in the queues can be a mixture of non-translatable 8-bit ASCII, translatable 8-bit ASCII (romaji), non-translatable 8-bit kana, translatable 8-bit kana, or 16-bit katakana/hiragana/romaji. As in the default line discipline, no state information is kept about the data in the queues or the `tty/jtty` attributes that were set when the data was input. For the translation of 8-bit data to 16-bit data, Kanji UNIX created an intermediate input queue for data needing translations. Therefore, all information about the data must be discerned from the input queue it is in, the data itself, and the current `jtty` attribute settings.

Backspacing is a good example. Assume an erase character is received as input.

1. If the romaji buffer is not empty then remove the last byte in the buffer, go to step 4.

2. If input translation queue is empty, then remove the last byte in the input queue. End backspace processing.
3. Remove the last byte in the input translation queue. If the byte removed from the input translation queue is the second byte of an EUC code, then remove the next byte in the input translation queue.
4. Move as many of the bytes off the end of the input translation queue into the romaji buffer that will make a legal partial romaji sequence. End backspace processing.

There were other issues associated with escaping metacharacters on multiple queues in multiple modes. Handling break and interrupt characters was difficult because of the interaction with the user level daemon process.

3.4 Raw Mode

The Kanji line discipline does not support any translations in raw mode. The notion of raw mode assumes one keystroke for each piece of data. This is not so when doing romaji-to-kana, kuten-to-EUC, or kana-to-katakana/hiragana/kanji translations. These translations require multiple keystrokes to enter one "character". However, kana entered directly from the terminal is supported in raw mode.

3.5 Kernel Data Structures

In order to maintain compatibility with System V, Kanji UNIX does not modify any System V kernel data structures. For example, Kanji UNIX needs to extend the type of information that exists in the `tty` structures. Because some parts of the kernel and some programs depend upon the size of the `tty` structures, it cannot be changed without causing incompatibility. Kanji UNIX got around this problem by creating new data structures, the `jtty` structures which exist in parallel with the `tty` structures. Since these two structures have a one-to-one mapping, it is simple to create a table which maps each `tty` structure to its corresponding `jtty` structure. When the system boots, the mapping is initialized.

Another approach to implementing Kanji UNIX may be to use the streams mechanism [RIC84] that will be available in future releases of UNIX. A Kanji streams module could be pushed into the normal `tty` stream.

3.6 User Interface to the Kanji Line Discipline

The kanji line discipline provides many features for the user and user level programs. Just as a user of a terminal uses the control characters in the default line discipline (i.e. the erase, interrupt, kill line characters) [Uni85], the terminal user is provided additional kanji control characters. These characters include:

- the **VROMAJI** character which is used to tell the line discipline to translate whatever is typed from romaji to kana,

- the **VHIRAG** character which is used to tell the line discipline to translate the preceding kana's into hiragana,
- the **VKATAK** character which is used to tell the line discipline to translate the preceding kana's into katakana,
- and others.

For user level programs, several **ioctl** are provided such as:

- **TCJGETA** will get the the kanji attributes associated with a given **jty** structure, **TCJGETA** will set the the kanji attributes associated with a given **jty** structure,
- **TCJLINEDISC** will return true if the kanji line discipline is active,
- **TCJDAEMON** will associate the calling process as the dictionary daemon for given **jty** structure,
- and others.

4. Implications for Other Native Language Supplements

The Kanji UNIX system was designed with the goal of being portable to languages other than Japanese. None of the kernel work was done using any special representation of Kanji UNIX.

4.1 Thai UNIX

As an example of the portability of Kanji UNIX, we were able to port and modify Kanji UNIX into a Thai version of UNIX in a matter of a few weeks.

The most significant difference is that several alphabetic characters form one written character and these written characters are displayed on a terminal as multi-row characters. This meant the method of displaying characters had to be changed.

Other more straightforward changes included modifying the parallel **tty** (in this case it was called **thtty**) structure for attributes required by Thai language terminals.

4.2 JIS and Shift-JIS

There are really two different information standards (code sets) for the Japanese language. JIS, and what is known as shift-JIS. While JIS is the preferred standard, Kanji UNIX supports both shift-JIS terminals and programs that speak only shift-JIS. This is implemented in the kernel fairly simply. In the routine that moves of data into (or out of) user space, just before the move, all data is translated to (from) shift-JIS. The same thing happens for a shift-JIS terminal.

5. Utility Considerations

Most of the standard UNIX utilities were unaffected by the Kanji UNIX system. However, several utilities needed to be modified to support various changes that were now visible to the user system.

5.1 Eight-bit Data

Some programs did not allow a full eight-bit data path. The shell (**sh**) for example used the upper bit in the bytes of some characters for “glob” expansion. This meant that neither EUC, nor kana characters could be typed into the shell, as it attempted to use the characters for various expansions. This was also a problem for **cu**, **pg** and several other programs. These programs were rewritten to avoid such problems.

5.2 EUC Data

Some programs that look at the data in a file needed to know about EUC codes. Because EUC codes must be considered in pairs, it is possible to find a match of one EUC code, with parts of two other EUC codes. This was a problem in **egrep**, **awk** and several similar programs. Once again, these programs were rewritten to avoid these problems.

5.3 Added Functionality

Several programs needed either to be written, or modified to support the added functionalities of Kanji UNIX.

- | | |
|----------------|---|
| shl | The shell layer manager had to be changed to support the jtty structure. This structure must be copied, and restored between each layer. Because shl itself works in line discipline number 2, the Kanji UNIX interfaces are not available from within the manager. The only restriction this causes is that layers may not have names in any of the Japanese representations. |
| vi | The text editor jvi had to be written to handle double width characters, JIS translations, screen handling, and related problems. The editor jvi was based upon the code of vi . |
| stty | The stty command was extended to handle setting the modes, and control characters specific to Kanji UNIX. The command works normally under the default line discipline, with no extra options available. However, in the Kanji line discipline the control characters, and translation modes are settable via the standard interface. |
| setterm | This program is used to setup a terminal for using Kanji UNIX. When used with a terminal whose name ends in “-j”, the terminal is switched into the Kanji line |

discipline, the appropriate escape sequences are loaded into the kernel (via `ioctl`s), and the "guide" line is reserved.

setjpr This program is similar to the **setterm** program, but is designed for printers. The basic function is to open the devices associated with the printer, switch the device to the Kanji line discipline, then to go to sleep, thereby keeping a last close from happening.

6. Issues for Standardization

Some of the issues currently being discussed in such fora as the `/usr/group` Technical Committee on UNIX Internationalization [us85] are:

Data transfer between heterogeneous language based systems

What does it mean to take a text file created under Kanji UNIX, and transport this file to a UNIX machine running Thai UNIX? How should the data be displayed? Should there be any conversion possible?

Code set size

How large does the code set need to be? To represent the entire Chinese character set would require a code set of 32-bits, while European languages only need 8-bits. Should both exist, or should there be one superset?

Concurrent code sets

How can a system store data from more than one code set in a single file? How can a user enter data from more than one code set in a given session?

Collating sequences

In Chinese, the natural sort order is by stroke count, while in Spanish the letter 'ch' follows 'c' and precedes 'd'. How can these languages be easily sorted in their natural order?

Directionality

Most European languages are read left to write; many Oriental languages however are read top to bottom, while Arabic and Hebrew are read from right to left. How can these natural orders be preserved?

7. Conclusion

While UNIX was originally a system designed with only ASCII in mind, only a few of the modules associated with the kernel needed to be added to extend the capabilities of UNIX to

support Japanese and Thai code sets.

As a result, in less than a year the Kanji UNIX product went from initial design to a released product. Kanji UNIX is a fully functioning UNIX system capable of supporting multiple language code sets, and is readily expandable to handle other natural languages representations.

BIBLIOGRAPHY

- [ATT86] AT&T UNIX Pacific, "UNIX System V Japanese Application Environment Release 1.0, Product Description and Installation Guide", January 1986.
- [RIC78] D.M. Ritchie and K. Thompson, "The UNIX Time-Sharing System", B.S.T.J. 57, No. 6 (July-August 1978), pp. 1931-1946.
- [JAP85] Japanese UNIX Advisory Committee, "Proposal to AT&T for Japanese Capability on UNIX* System V", April 1985.
- [RIC84] D.M. Ritchie, "A Stream Input-Output System", AT&T Bell Lab. Tech. J., 63, No. 8 (October 1984), pp. 1897-1910.
- [SCH86] J. Schriebman, "International Application Portability", outline for UNIX HUI-86 New Zealand talk, March 1986.
- [UNI85] UniSoft Systems, "UniPlus+ System V, Release 2: Administrator Manual", 1985.
- [YOS86] I. Yoshida and R. Jung, "Specification of vi for the Extended UNIX Code", internal report generated for AT&T UNIX Pacific, January 1986.
- [US85] /usr/group Technical Committee on UNIX Internationalization, "Issues in International UNIX", working paper, December 1985.

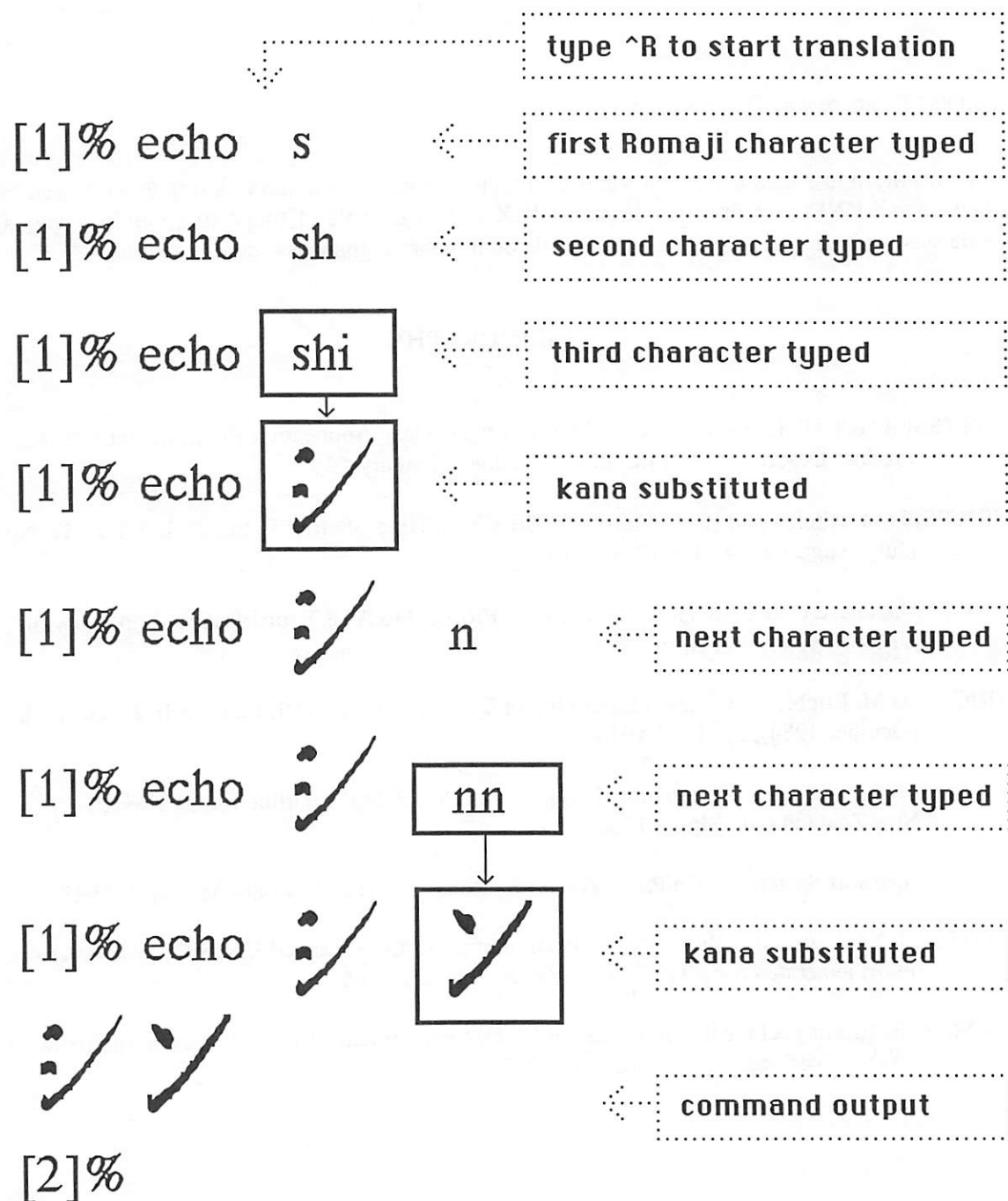


Figure 1: Romaji to Kana Translation
How a command line is successively altered

(Kana's are not drawn to scale.)

[1]% echo シン ← type ^E to translate to Kanji

[1]% echo 新

新

[2]%

R E 1/20 1. 申 2. 神 3. 信 4. 心 5. 新 6. 辛

Figure 2: Kana to Kanji Translation
Dictionary Look-Up Daemon

(Characters are not drawn to scale.)

[1]% echo シン from **Kana**, type:

[1]% echo シン ^K to get **Katakana**

[1]% echo しん ^G to get **Hiragana**

[1]% echo 新 ^E to get **Kanji**

Figure 3: Kana to Katakana, Hiragana,
and Kanji Translations

(Characters are not drawn to scale.)

Tools for the Maintenance and Installation of a Large Software Distribution

D.M. Tilbrook
P.R.H. Place

Imperial Software Technology

ABSTRACT

This paper describes the problems inherent in developing and maintaining a large software distribution. A strategy for software development and its relevance to these problems is discussed. Brief outlines of policies that implement the strategy are then presented. The UNIX¹ implementation at IST is described with examples of the support tools developed. Finally, more detailed descriptions of some tools are presented, with particular reference to *pmak*(1), a front end to *make*(1).

1. INTRODUCTION

In general, programming techniques applicable to small scale software projects do not scale up to large, or even medium scale, projects. At Imperial Software Technology (IST), we are developing software (approximately 3,000 files, 300K lines, 7.2M bytes of source code) in two projects: ISTAR, a project developing an integrated project support environment, and 7001, a project researching into software development, described in this paper. The continuing development, application and distribution of the 7001 software has highlighted some of the problems inherent in software development. In an attempt to resolve these problems, an overall software strategy has been evolved and adopted. Further, particular policies for carrying out this strategy have been developed, both in the form of standard practices for the development of software, and as tools to assist that development.

2. PROBLEMS

The development and maintenance of 7001 software has enabled us to identify a number of problems that occur in large software projects. We describe these problems and also give an indication as to why *make*(1)², as used in the past is inadequate for large scale projects.

2.1 Education

If a company is to adequately assure the quality of its products, then it needs to impose a company style of coding on its programmers. This requires education in both coding standards and available tools. Unfortunately, when projects have to deliver products in short time scales, this education is often thought to be a luxury that may be omitted. Which in turn leads to a diversity of style in code developed for a client.

2.2 Interdependency of Construction

Software often contains complex dependencies, where the construction of a program is dependent upon the prior construction of other programs or libraries. It is not an easy task to specify in a clear and precise way the order in which libraries and programs should be constructed.

A particular example of this is the mail system, *ma*(1), used at IST (a system which rivals the Rand mail handler in complexity - but **not** in code size). The construction of *ma*(1) depends

1. UNIX is a trademark of AT&T Bell Laboratories.

2. The term "make" will be used extensively throughout this paper as a verb, a proper noun and an adjective. In each case, its use should be clear from the context. Therefore most occurrences will not be distinguished in any way.

upon the prior installation of *arlo*(1) which in its turn depends upon TIPS, TIPS requires *xdb*(1) which is dependent upon the IST library.

2.3 Application of the Software

We use the 7001 software in three different ways: As a research tool, as a production system and as part of a product, ISTAR. This entails the maintenance of three versions of the software: the research version which is updated on a continuous basis; the production version, used within IST and updated every two or three months; and the distribution version, delivered to customers and updated at longer intervals. This creates a problem of maintaining consistency across the different versions of the software (a problem that is only partially solved by considering the research version as the master copy).

2.4 Identification

When a number of people change the source code, there is little stability in a system and a piece of software can suddenly stop working because a change to some other piece of software has been made. We have to be able to keep track of these changes in order to determine responsibility for code and to maintain control over development.

2.5 System Variations

At IST, we use a number of different machines running different versions of UNIX. Specifically, we have VAX/750's³ and a 3B2 running System VR2, as well as a number of 68000's running Uniplus System V, a Pyramid 90 running OSx and a VAX/750 running 4.3bsd.⁴

With this variation in machines and operating systems, we have had to face the problem of "standard" tools varying between machines, (e.g., *install*(1) and *lint*(1)), files being installed in different directories (e.g., *lint*(1) and *wait.h*), files having different names (e.g., *string.h* vs. *strings.h* vs. nothing at all), routines having different names (e.g., *strchr*(3) vs. *index*(3)), and routines returning values of different types (e.g., *sprintf*(3)). One previous solution to this problem has been to overload make, however, this often results in very complicated make scripts.

2.6 Problems with Make

Whilst the problems described above are significant, the most difficult problems faced are those caused by make. Make is probably the most important tool in an installer's tool-kit and there can be little doubt that without it, or some tool with a similar function and power, the installation of a large software system would be almost impossible. Most of our efforts have been directed at the development of tools to use make more effectively, not to replace it. It is our opinion that make should not be enhanced, but limited in power in order to achieve improved performance and to eliminate features that interfere with its more effective use, which is to say, make make make things well. However, make does have a number of short-comings, which we now describe.

2.6.1 Differences Between Versions There are at least three versions of make in use. For the most part the fundamental syntax and function are the same. However, there are substantial differences with respect to the handling of construction rules and archive files.

2.6.2 Lack of Standards Despite the use of make for about ten years, no dominant style of use has evolved. There are a number of target names that are frequently used in make scripts (e.g., *all*, *clobber* and *clean*) but the associated functions vary from script to script. Comprehensive make scripts tend to be exceedingly large and complicated. A major cause of complexity in make scripts is the tendency for programmers to overload the script with rules and operations not

3. VAX is a trademark of Digital Equipment Corporation

4. We have also run the 7001 software on 4.1bsd, 4.2bsd, Ultrix, System III and V8.

concerned with construction.

2.6.3 Duplication of Information Make scripts often have information duplicated (in slightly different forms) in different parts of the script. For example, a single file name may appear, in various forms (e.g., *echo.c*, *echo.o*, *echo*, */bin/echo*) throughout the script. File names often appear in lists and the ordering of these lists may be significant, however, it is not possible to determine the significance of the lists without examining the entire make script.

A further cause of duplication of information is that the current form of make forces a conscientious software designer to give the name of an object library file at least twice. Consider a make script to compile a program that depends on a library, then the library name should be given in the dependency list for the program and also in the compilation command. This requirement for the multiple entry of information leads to either incomplete, inconsistent or redundant specifications.

2.6.4 Weakness of the Time-Dependency Relation One cause of information duplication is that the time-dependency relation is difficult to limit, non-transitive and inadequate.

To construct a target one is forced to make targets depend on intermediate files which in turn depend on the source files. But there is no way to express the dependencies such that unnecessary constructions are avoided if the target is up to data with respect to the source, even though the intermediate files don't exist.

A further problem with the T.D.R. is the propagation of redundant operations due to changes in a file's modification time but not its contents. This is particularly a problem when databases are used to store information that is transformed into large numbers of intermediate files⁵.

2.6.5 The Remake Problem Another problem with the T.D.R. is that the only way to force the reconstruction of a target is to remove it or to *touch*(1) a file on which it depends. The IST software uses a number of binary representations that are generated by processing a textual representation. If the data file depends on its generator (as well as its source) then it will be regenerated every time the generator is changed, even when the change is due to a cosmetic or unimportant modification. But if the data file doesn't depend on its generator there is no way to force its recreation when it is necessary (e.g., a change in the encoding). But there is no convenient or commonly used mechanism within make to handle such a problem.

2.6.6 Difficulty of Make Script Maintenance Because, as a consequence of the above problems, make scripts are often very complicated, changing them is a difficult and error-prone task. Due to time constraints or lack of concern, make scripts tend to be "hacked" and not designed. Additions are made by duplicating lines that are "close" to what is required and then making necessary changes which leads to unstructured, over-complicated and incorrect scripts.

2.6.7 Dynamic Dependency A product may depend upon some dynamic list of data files. An obvious dependency to use is: "target: *.d". However, although make will correctly rebuild *target* if a new data file is added, it will fail to rebuild *target* if one of the data files is removed.

2.6.8 Include Facility There is no universal or useful include facility. The standard make on *bsd* distributions does not support an include facility. The AT&T versions have a form of include that is rendered inadequate by requiring the include parameter to be an absolute pathname (i.e., there is no search path and variables may not be used to resolve the location of the desired file).

It is essential that make scripts be able can include files to establish environmental settings and have a search path that can be externally specified.

5. The database that describes the *arlo* keywords is processed to produce over 50 header files yet a change in the data base may only change the contents of one of those files.

2.6.9 No Conditionals The lack of conditional tests within make means that it is not easy to dynamically select (or suppress) a given construction. Due to syntactic problems, using the shell to perform tests is both a frustrating and difficult exercise. It is also a non-trivial task to ensure that errors are handled correctly. For example, one has to ignore the result of any test that might legitimately return a non-zero status, but this means that real errors aren't handled properly.

2.6.10 No Local Assignments It would be desirable to use make's limited variable facility to be able to design and use prototype constructions. Unfortunately, because make assigns values to its variables in the first pass and uses the values in the second pass, only the last assigned value will be used.

3. SOFTWARE STRATEGY

In order to solve the problems described in the previous section in an effective manner, a software strategy has been evolved. The main aim of the strategy was to reduce the difficulty of developing and maintaining code. A second important aim was to make it easier to follow, than to ignore the strategy.

3.1 Isolation

A primary assumption is that the software is to be installed on a "vanilla" UNIX system, which is to say that none of the bugs have been fixed and no enhancements are available. Thus, the software installation process must make minimal assumptions about its environment. Another consideration is that the software should be totally isolated from the rest of the system. It should be possible to install the software, test it and remove it without having affected the original system. It should also be possible to ensure that such a test does not use previously installed or enhanced parts of the environment.

3.2 Provision of Tools

As has been stated, many of the tools available under UNIX vary from system to system or have bugs which make them less than helpful. Where necessary, distributions must include tools to rectify known bugs or to provide missing facilities.

3.3 Single Sourcing

Another important part of the strategy is single sourcing of information. If information is expressed in only one location, then there is a much greater chance of that information being in step with the system. Therefore, all the site dependent information should be contained in a single location and processed to create the relevant files. Where possible, information about constructions or installations should be stored in a single well known location. For example, we use a file called *system.h* to contain the version of UNIX being used. All other files that refer to the version (or name) of the system use this file.

A further example of single sourcing is the embedding of TIPs database entries in the C source code of the library functions. These TIPs entries are extracted from the source and are used to generate *lint(1)* libraries, *xdb(1)* databases (used to provide online glossaries and references), the software inventory database entries, and the standard manual sections. Thus the source file for a routine contains all the necessary information and is used to create all the relevant products.

3.4 Disciplined Organisation of Source Code

A further feature of the strategy is to separate files by their use (or function) into different directories. This leads to a software organisation where only one library is constructed from each directory and where shared header files are placed in separate directories.

For example, we avoid files with the name **main.c**, the reason being that **main.c** does not help identify the function of the program, whereas *passwd.c* suggests that the file is the main source file for *passwd(1)*.⁶

3.5 Locatability

Another aim is to be able to easily locate the source for any program. Therefore, we must construct a source inventory database as well as tools to support its construction and interpretation.

3.6 Self Documenting Programs

In response to the problem of training new personnel, programs should be substantially self-documenting. Since manual entries often provide too much information, at the wrong level, in the wrong order and, perhaps, out of date, help information should be generated from, and incorporated into, each program.

3.7 Reducing the Complexity of Makefiles

In order to resolve some of the problems with make, we should reduce the complexity of make files, or even render them unnecessary. Tools must be provided to support the creation and maintenance of make files and to provide alternatives to make for non-construction operations. The existence of a source file in a directory should be sufficient indication that the file is to be used. Information about a program's construction should be stored in the program source, not in the make script.

3.8 Auditing

In order to solve the problem of traceability, source code control techniques should be used as a default action, rather than as an afterthought. Each time a file is changed (or installed), a record should be kept of the reasons for the change, the person responsible for the change and the change itself.

SCCS or a similar system, is necessary to perform this record keeping. However, we also require that a record of any changes be kept in a form that may be used by other tools. In order that this record accurately reflects changes, the record keeping should be automatic, requiring no effort from the user. Further, each time a program is installed, a record should be kept of the installation, so that all the changes to a system are recorded. Again, this record-keeping should be automatic. Tools are then needed in order to create and manipulate these records.

4. POLICIES

Over the last decade, subsets of the current 7001 software have been installed on a variety of machines including V6, V7, V8, PWB, 4.1bsd and System III. In general, these installations were prompted by changes of hardware or operating system. The early installations were manual, requiring a large number of changes to the source and took weeks to complete (due to other higher priority problems). In fact, it was the complexity of installing the software that eventually led to the work described in this paper. More recently, (circa 1982) attempts were made to develop personal techniques and coding styles that would reduce the number of problems faced in future installations. In 1984, it became necessary to simultaneously maintain the software on two different environments (4.1bsd and System V). Furthermore, the software was not to be installed on the System V machine by its creator. Thus the evolution of a comprehensive software strategy became of utmost importance.

4.1 The D-Tree

One of the foremost problems with the early distributions was the creation of the magnetic tape containing the source, the whole source, and nothing but the source. Practically every

6. There are 53 files called main.c in our 4.3bsd source trees. Yet there is no adb.c, make.c, lex.c, yacc.c and the only sh.c is source for the program installed as csh.

installation prior to 1984, was delayed by the absence of necessary files. To resolve these problems and to achieve the isolation discussed in the previous section, it was decided that all source files would reside in a single tree, henceforth referred to as the "D-Tree". All installed files would also reside in a single tree, henceforth referred to as the "\$USRIST" tree. A single setting specifies the default location of the \$USRIST tree.

Furthermore, changes to the "base" system have been avoided when possible or, when the changes were unavoidable, installed in a non-standard location. This means that the "D-Tree" installation can be tested on a "vanilla" machine by excluding the updated utilities from the search path.

4.2 The Magic Directory

Another major problem faced in any installation is the establishment of the site, machine, system and configuration dependent information. In addition there are number of files required on certain systems but not on others. Or a choice has to be made amongst a set of files that implement a particular facility that varies between systems. The main mechanism for handling this variation is to isolate the control information to a single D-Tree directory, *magic*. For a new installation, the *magic Makefile* needs to be modified, and "site" and "machine" files created. The change to the *Makefile* is to specify the version of UNIX being used, the default \$USRIST path and the names of the site and machine files. The site file specifies information about the company and location (e.g., address, phone number) and will be normally be the same for all installations of the software within the same company. The machine file sets controls for the specific installation (e.g., the make pathname and the mechanisms to be used for archives, default owner and group of installed files and attributes of some of the standard installation tools).

In addition to the *magic* directory, the D-Tree, and similar distribution trees at IST usually contain a further seven directories: 1) doc - reference documents and tutorials. 2) hdrs - the source header files. 3) install - shell scripts to do phased and environmentally pure installations. 4) man - the manual section tree. 5) src - the source for the distribution. 6) tools - contains a raw ("vanilla" to the extreme) make script to create the minimal set of tools needed to install the D-Tree and, 7) FL - file list maintenance system

The *FL* directory merits particular attention. The combination of the \$USRIST and source tree normally consists of 33 Mega-bytes, 500 directories and 6000 files. In addition to the 3600 source and SCCS admin files, there are binaries, object files, object archives, generated header files, generated pmak scripts, diagnostic outputs, test data files and programs, ctags files, SCCS p-files, and the occasional file called *core*. To ensure the integrity of the distribution file list it is essential that all necessary files are properly administered and registered. The FL subsystem is used to ensure that the system is "clean" and complete.

4.3 Program Source Organisation

There are a number of conventions that dictate the format of standard header information in each file, as well as the manner in which files are organised in the tree.

- All files within a single directory are "of the same type". For example, the files that make up the library routines for some program will be stored in a directory separate from the directory containing the data files. Similarly, header files will usually be in another directory.
- All files containing code have an SCCS identification string. Thus it is possible to examine a program's binary and determine which versions of the source code were used to construct the program.
- Each main program has a special comment line which indicates the libraries to be compiled with the program. This line, the LIBS() line, has a number of important uses (described later) and is fundamental to the operation of a number of tools.
- So that programs are self-documenting, a program can always be executed to retrieve its usage. To achieve this aim, every program written at IST can be executed with a **-x**

(explanation) flag. Executing a program with the “-x” flag will print its usage line and brief explanations of the program and its flags. In some cases, where the input to the program is in a fixed form, the -X flag is used to describe the input format.

- We employ a number of file naming conventions for two major reasons. The first is to distinguish between true source and temporary files. The second is for the recognition of the type of a source file so that the *pmak*(1) script generators can find required information and generate appropriate constructions.

4.4 Tools

In this section, we will describe a few of the tools (there are many others) in the D-Tree.

4.4.1 *instal* The reason behind the construction of our own version of *install*(1) is that versions vary or may not be available and we need the enhancements as described below:

- *instal* flags are specified in three ways: the environment variable “\$INSTFLAGS” (used to specify global settings such as default owner and group); the file *./Instflags.L* (used to specify special cases for specific files such as setuid modes and/or ownership selectable by system name); and on the command line (rarely used).
- *instal*(1) provides the entry for the audit trail automatically. The importance of the audit trail has already been stressed and we believe in the importance of its construction with “no cost” to the users.
- *instal*(1) enables a software developer to install a program whilst the program is still in use.

Instal(1) has 21 flags, even more than *bsd*’s *ls*(1), but one of them is -x which displays a brief description of each flag. However, it is very rare to explicitly use any flags when invoking the program since they are usually specified via \$INSTFLAGS or *Instflags.L*.

4.4.2 *sccs* We use Eric Allman’s SCCS interface, supplemented with operations to remove and rename files and to record all changes (e.g., delta’s, renames, removes or *rmdel*’s a file) via *dmail*(1).

4.4.3 *dmail* This program goes up through the directory hierarchy until a **Dmail** file is found, at which point it appends a record of the change to the Dmail file. Whenever a distribution is created, a comment is added to the Dmail file. Thus all changes to the system after a distribution are easily identified.

4.4.4 *mkdist* This program uses the *dmail* format lines to create an update tape. It can extract versions (as identified by *sid*) or current or obsolete source files.

4.4.5 *filelist* This is a tool that maintains a dynamic list of files and a collective last modification time. That is to say, given a file *flist* containing a list of files, *filelist*(1) may be used to set the modification time of *flist* depending upon the file list contained. This program helps solve the problem of a product depending upon a changing list of files.

4.4.6 *cpifdif* This tool compares two files, and if they differ it copies the first file onto the second. *cpifdif*(1) is used extensively in the software construction process in order to minimise the amount of unnecessary work (it helps to circumvent one of the make time-dependency problems). *cpifdif*(1) is approximately 25 times faster than the equivalent shell command, does not require invoking a shell and exits with a meaningful exit status.

4.4.7 *incls* *incls*(1) files for “#include” lines and other such directives and outputs the appropriate pathnames in a variety of formats, including a make dependency list.

4.4.8 *com* *com*(1) was (initially developed by Tom Duff) extracts a line of the argument file that contains the string “/*%” or “/*@” and executes the rest of that line as a shell command after replacing any embedded “%” or “@F” by the argument file name. The “/*@” string may contain a number of other directives including: “@I” (replaced “-I...” arguments as specified by \$INCLPATH) and “@L” (the converted “*LIBS:” line).

5. PMAK

Whilst some problems were solved by the policies and tools thus far discussed, most had to be solved by the development of better techniques and tools for the generation and interpretation of make scripts. The major result of this effort was *pmak*(1)⁷.

5.1 The History and Evolution of Pmak

The initial research at IST on the development of better techniques to control software construction and maintenance was done by David Tilbrook and Paul Parker in 1983. In the beginning, this effort concentrated on separating the specifications of: 1) the relationship of components (analogous to a make dependency list); 2) meta-operations (make constructions not bound to specific types); and 3) type transformations resulting from bound meta-operations (the cross product of the first two parts).

This approach was taken because it was felt that some of make's problems were due to its inherent dichotomy: it tries to be both a database and a command processor, without employing a proper supporting technology. In our prototype, we tried to avoid this problem by using Prolog⁸. Our conclusion was that a weakened form of the approach was semantically valid, however, the syntactic and performance problems were prohibitive⁹ and the work was abandoned. The most important contribution to our current work was to discourage any objective other than software construction and installation.

Initially, *pmak* was a shell script that interpreted the argument list, used the C language preprocessor (*cpp*) to preprocess the input script (primarily to provide a proper "#include" facility), and then invoked make to interpret the output. The *cpp* style of conditionals and macro assignments was used to deal with the differences in the versions of make and operating systems.

The next development was the construction of a simple script generator to handle the multi-directory make problem. This allowed us to develop a single, top level controlling make script which had a simple update and extension mechanism. Other script generators followed to handle archive library and command directories. These developments were facilitated by the previous development of *instal*(1) and *incls*(1).

It soon became obvious that *cpp* was inadequate¹⁰, non-standard, and contained numerous bugs. Another problem is that there is no method of commenting the input that is acceptable both to *cpp* and make. Furthermore, while the initial use of *cpp* solved some problems, it was apparent that some additional built-in facilities were desirable, to improve or facilitate the handling of the -I arguments to *cc*(1) and library pathnames. Thus *mpp*(1) was created to handle comments properly, to be software that we could distribute and to provide some additional facilities such as "#elif", expressions to test for the existence of files and built-in macros "LIBS(file)" and "InclFlags()".

Attention was then turned to the *pmak* shell script. At this time, the complete construction and installation of the D-Tree was performed by a single *pmak* script that invoked *pmak* scripts in sub-directories. However, this used an excessive number of processes (it could wrap the process ids easily) and an excessive amount of time. *Pmak* was being executed 300 to 400 times per D-Tree installation. A version of *pmak* written in C was created that improved performance considerably. Also, enhancements were being made to *mpp*(1) to deal with the needs of the evolving script generators such as the addition of facilities to handle *cc*(1) "-I" flags and shell

7. In footnote #2 "1,\$s/\\(mak\\)e/p\\1/g".

8. There are people who claim that Prolog can do this!

9. Is the cray free? I need to reinstall */bin/true*!

10. One minor problem was that *cpp* ignores anything after "/*" up to the next "*/". The *cpp* "-C" flag (preserves comments for *lint*(1)) was not sufficient since the only effect was to output the comments literally.

environment variables.

During the development, it became obvious that *mpp* could (and had to) be incorporated into *pmak*, primarily so that environment variables could be set in the *mpp* phase and exported to the make process and to support the specification (within the input) of other application controls (e.g., the actual pathname of the application).

By this time, most of the directories contained *pmak* scripts that invoked a script generator to create a subscript (if it didn't already exist) and then execute *pmak* on the subscript. This was expensive and inconvenient. Therefore a new mechanism, the *pmak* control file (known as the PMC file), was created to specify the arguments to the required script generator and *pmak* was modified to recreate the temporary script, *Pmak_*, whenever it did not exist, was older than the PMC file or the *pmak* "-s" flag was specified. At this time nearly all the *pmak* source scripts were replaced by much shorter and simpler PMC files.

The current distribution contains five make scripts (all part of the *magic* installation and initial bootstrap which are therefore used before *pmak* is built), some special case *Pmakfiles*, and 124 PMC files averaging 4 significant lines each.

5.2 The Pmak Processing

Pmak is a process that controls and prepares input for other processes. Its major role has been as a front end to make, but it can be and has been used for other applications. Its processing can be split into three distinct phases:

- script selection and generation (PMC processing)
- script processing and expansion (*mpp* phase)
- invocation of the application (e.g., make)

The first phase is perhaps the most interesting and novel aspect of the *pmak* system and enables us to achieve, with very few lines of information, a high degree of control over the construction process. Two of the five standard script generators will be discussed in later sections. The second phase is functionally similar to *cpp*, but it does exhibit a number of important facilities that will be described in the next section.

5.3 Pmak Preprocessor Controls and Directives

The primary role of the *mpp* phase is still to provide the "#include" construct so that environmental controls and prototype make scripts could be used easily. The following features are either novel or extremely important:

5.3.1 #inherit In addition to the "#include" facility, which is used to establish and set system wide controls, it is often desirable to override some of those settings for a particular sub-tree. For example a user altering a part of the distribution copied into their \$HOME tree may wish the programs to be installed in \$HOME/bin without changing any of the construction control files. It was for such cases that the "#inherit" facility was invented. It is similar to the "#include" facility but differs in an important way. For the directive "#inherit Lcl.vars", all files called *Lcl.vars* from the root ("/") down to the current directory will be included, and in that order. For example, the files */dt/Lcl.vars* and */dt/src/games/Lcl.vars* both exist. The above "#inherit" directive will cause both files to be incorporated into games directory scripts with settings in the latter overriding the values in the former. In general, this facility is used to establish necessary values (e.g., \$DESTBIN) at the top of the tree but to provide local overrides, if necessary.

5.3.2 LIBS() In order to implement the strategy of single sourcing and to provide a better mapping between libraries and path names, the *LIBS()* macro was introduced. The occurrence of the string "LIBS(src.c)" in *pmak* input is replaced by a string based on the first line, found in the file *src.c*, that contains the string "*LIBS:". When such a line is found, everything up to and including the "*LIBS:" is removed. Then, any words of the form "-IX" are converted into a string provided by *pmak* (i.e., */usr/lib/libcurses.a*), a string provided by the user mapping file, or

to the full pathname of a file, *libX.a*, found in the normal library search path (may be extended using “\$LIBPATH”).

Sufficient emphasis cannot be placed upon the importance of this mechanism. The single sourcing policy is achieved since there only one legitimate place for the list of libraries, used by a program, to occur and that is at the start of the source file that contains “main()”. Hence, any other programs that use the list of libraries will be using the correct values.

Using this facility, the make script dependency list for a binary contains full path names for the libraries used and the compilation statement uses the same list. The string “LIBS()” (i.e., no argument file) uses the last string generated, since the compilation statement frequently follows the dependency lines as in:

```
echo:          echo.o LIBS(echo.c)
              $(CC) $(CFLAGS) $@.o LIBS()
              mv a.out $@
```

Other programs have been modified or created to use the “LIBS” line. *com(1)* replaces “@L” in the command line by the transformation of the argument file’s “LIBS” line, and a front end to *lint(1)* has been created that extracts the “LIBS” line and does a similar conversion to the appropriate lint libraries and appends the result to the end of the argument list.

5.3.3 Touch() In the section on make we discussed the problem of regenerating a target when the cause is not a modification to any of the files in target’s dependency list. This problem cannot be solved without changing make; we have created a mechanism that provides a limited implementation of the desired facility.

The macro “Touch(prog)” is replaced by the pathname for a file *touch/prog.h* in the “#include” search path. If no file with this name is found, then the macro is replaced by the null string. For many types of target file, “Touch(prog)” (where *prog* is the tool that creates *target*) appears in the dependency list. For example:

```
target:        target.src Touch(prog)
              prog -o $@ target.src
```

If *touch/prog.h* does not exist or is older than *target* then the “Touch(prog)” has no effect. But, if we wish to force the reconstruction of all targets produced by utility *prog*, we simply *touch(1)* the *touch/prog.h* file in the search path. This mechanism is used to solve the re-make problem discussed in section 2.6.

5.3.4 IfOlder() The “IfOlder()” macro is used extensively to express dependencies in a way that can avoid the propagation of unnecessary constructions and processing. The argument is a white space separated list of files. The second and subsequent files in the list may be prefixed with by “!”. To explain the semantics consider the following example:

```
IfOlder ( f0 f1 !f2 )
```

This string will be replaced by *f0* (i.e., the first file in the list) if file *f0* does not exist, or if file *f0* is older than file *f1*, or if file *f2* does not exist. Otherwise, the macro is replaced by the null string.

The following is an example of IfOlder use in make script and is representative of every combination of lex and yacc files in the distribution¹¹.

11. The explanation of why and how this eliminates unnecessary recompilation of *Fl.c* is left as an exercise. It appears in the EUUG proceedings in full but was removed to meet USENIX imposed limitations.

```

Fy.c y.tab.h:  Fy.y
               $(YACC) -d Fy.y
               mv y.tab.c Fy.c
               cpifdif y.tab.h Fy._h
Fy._h:  IfOlder ( y.tab.h ! Fy._h Fy.y )
Fl.o:  Fl.c Fy._h
Fl.c:  Fl.l

```

Readers might be quick to find fault with this approach since it seems to be far too complicated to understand and use. Indeed, we would agree with them if we had been forced to write such a construction. However, the pmak script generator *pmcmd(1)* created the above construction simply because the files *Fy.y* and *Fl.l* exist, which brings us to the final aspect of pmak that will be described, script generation.

5.4 PMC files

When pmak is invoked without an input script being explicitly specified, it searches the current directory for files called *Pmakfile* and *Pmak._*. If *Pmakfile* exists, it is used as the input script. Otherwise, if *Pmak._* exists and is newer than the *PMC* file, it is used as the input. Otherwise a new script is generated and written into the file *Pmak._* which is then used as the input script. The pmak “-s” flag forces the *Pmak._* file to be recreated thus may be used to override the comparison of the *PMC* and *Pmak._* last modification times.

Script generation is controlled by the pmak control file *PMC*. This file contains a command to invoke a script generator and frequently contains the input to the generator. The command is expressed as a *com(1)* line in the *PMC* file. For example, a *PMC* file in a directory of shell scripts and C programs might contain:

```

#/*@ pmcmd -f @F
# input to pmcmd

```

Normally the command will be one of the standard script generators, however, any arbitrary shell command may be used. Currently all 124 *PMC* files in the distribution use one of the standard generators.

5.5 Pmak Script Generators

Currently there are five standard pmak script generators, whose functions are as follows:

<i>pmcmd</i>	to process and install xdb databases and arlo scripts, to install shell scripts, to compile and install C, Yacc, and Lex programs.
<i>pmdirlist</i>	to control and invoke multi-directory makes (pmaks actually).
<i>pmlib</i>	to build and install either a library or a program built from the library.
<i>pmlfiles</i>	to process and install data files and to create directories.
<i>pmproto</i>	to create constructions for pmak scripts according to the given prototypes. The constructs created may be dependent upon file suffices.

5.6 Pmcmd

The primary purpose of *pmcmd(1)* is to process all C, Yacc, Lex, As, Arlo, Xdb, Mkhlp and Sh source files in the current directory by outputting a pmak script that will build and install the products in the appropriate location. The resulting scripts will contain **all** the dependencies for the products by using *LIBS()* for libraries, *incls(1)* to generate the “#include” dependencies for C and Xdb files, and *arlouses(1)* to generate a list of Arlo sub-scripts. The Arlo and Xdb dependency lists include the appropriate “Touch()” string as described previously.

The first phase of *pmcmd(1)* processing is to create a list of source files. For a file *prog.X*, *prog* is assumed to be the name of the installed product. For example, *src.sh* is the source of the installed shell script *src*. Ambiguities are resolved through built-in knowledge of the production

steps. For example, if both *file.y* and *file.c* exist, it is assumed that *file.y* is the source and *file.c* is the *y.tab.c* file (i.e., the yacc output).

The second phase is to read the pmcmd input (if any) which may contain directives to specify a variety of controls and overrides.

Rather than explain each directive in turn we will give a contrived and annotated example of a pmcmd input file. Comments are given after "--".

```
##/*@ pmcmd -o -f @F          -- PMC command

%if unix5.?
D dress                        -- don't build or install dress
%endif
d localtool                   -- don't install localtool
L vi ex                       -- to install vi link to ex
C par pary parl               -- par created from par.o, pary.o and parl.o
B init /etc                   -- install init in /etc
S bizarre                     -- suppress construction commands for bizarre
!                             -- balance of file output literally
# construction for bizarre
```

It is recognised that the directives are overly terse and could be improved substantially. There are also minor inconsistencies with the other pmak script generators that will be resolved eventually. It must be remembered that this is largely a research project and the pmak system is fundamentally a prototype. Despite these problems, most users learn to both use and understand PMC files relatively quickly, aided by the fact that all the script generators have "-X" flags that output a description of the input syntax and semantics. Most pmcmd PMC files are very small and the poor syntax is not a significant problem.

5.7 Pmdirlist

One of the major problems to be solved was creating a convenient and easy to use mechanism that could invoke constructions in multiple directories.

When dealing with hundreds of directories, it becomes very important to be able to select subsets for construction, express the dependencies and reduce the overheads involved in changing to a new directory and invoking another pmak.

The pmak script generator *pmdirlist*(1) was developed to process a list of directories and to produce a pmak script that could be used to invoke the required operations in selected subsets of those directories. As in the previous section an annotated example of a typical "PMC" file will be used to explain and illustrate *pmdirlist*'s input and features.

```
##/*@ pmdirlist -c IL -f @F
hdrs    I                    -- hdrs directory to be installed but has Instal construction only
+ -- "+" lines split the input into levels
lib     L      hdrs         -- lib depends on hdrs
cmd     ~      libndir lib -libist -- see next paragraph
```

The "~" indicates the default constructions are used. The "-" prefix specifies dependency on either Instal or Local construction of *libist* directory (which ever is appropriate at invocation). If a prerequisite directory is suppressed (due to system selections) or the non-existence of directory, it is ignored.

Explaining *pmdirlist*'s output is a formidable and profitless task since there are approximately 10 lines of output per directory and 40 lines per level. We will illustrate a tiny segment below, but for the most part it is not intended that *pmdirlist* output be read by humans or programmers.

The requirement that subsets of the constructions may be selected creates a large quantity of output. For example, nearly every directory in the distribution requires that the "hdrs" has been

previously installed. But there are rarely any "hdrs" constructions to be performed and confirming that this is the case is an expensive and generally unnecessary process.

To support the above requirement, pmdirlist creates pseudo Instal and Local constructions for each level and directory, as well as additional macros, used in dependency lists. To illustrate, if pmdirlist processed the following:

```
cmd    I    dist/tools lib
```

where *lib*, *dist/tools* and *cmd* are in the first, second and third levels respectively the following would be produced:

```
#if          LVL >= 3 && Ito < 3
Local:       Local3
Instal:      Instal3
#endif

Instal3:     cmd.I
cmd.I:       I1(lib) L2(dist/tools)
             CdMake(cmd) PmakFlags MakeFlags Instal

Clean::      ; CdMake(cmd) PmakFlags MakeFlags Clean
```

In pmdirlist output, "LVL" is used to specify the level up to which constructions are performed, and "Ito" is used to specify the level to which installations have been completed¹². At run time their values are used to select or suppress sections of the script and to define the macros that are used in dependency lists¹³. The user can thus invoke pmak to process individual directories, levels, and/or constructs via the command line, while suppressing the reprocessing of lower level directories. These mechanisms might appear to be complicated, but users who have to deal with large distributions employ them extensively.

6. USE AT IST

The techniques described above are in use at IST. They have been applied to the tools themselves and also to the ISTAR project. Programs imported into IST are reorganised into the 7001 organisational style or modified as necessary to take advantage of the pmak package. This is usually a straightforward task which makes any subsequent development of the software easier, if only at the level of re-compiling programs after bugs have been fixed.

Perhaps the best indication of the power and usefulness of these techniques is displayed by the D-Tree (wherein the tools described above are contained). There are 1400 lines of PMC, Pmakfile and Makefile¹⁴ in 139 directories that install about 900 program and data files. Further, even though we are maintaining the source on four different machines, the PMC files are identical. Compare this with */usr/src/usr.bin* on 4.3bsd, where there are some 1800 lines of make scripts with many of the prerequisites either missing or inaccurate, in 29 directories, installing less than 80 programs on only one system with no overall controlling mechanism.

12. The default values for "LVL" and "Ito" are 100 and 0 respectively. 32767 would have been a more logical choice for the former, but 100 is more than sufficient. The maximum number of levels used thus far is 9 and that was to handle 50 directories. The SDI project will just have to use some other system, manually specify the setting or change *pmdirlist.c*.

13. If "Ito" is less 1, I1(lib) is converted to "lib.I", otherwise it is converted to null string. Thus if Ito>=1 the dependency of "cmd" on "lib" is ignored.

14. All of the Makefiles (300 plus lines) are used in the initial bootstrap of the software only.

7. CONCLUSIONS

It would be nice to say that we are confident that the installation of the D-Tree on any “sane” UNIX system will be trivial, but we cannot. Practically every installation on a new system has raised some problem. However, experience leads us to believe that the modification of the D-Tree to take these problems into account is a relatively simple task. Most portability problems can be dealt with within the existing policies and conventions. Furthermore, by virtue of the number of systems on which the software has been installed, the frequency of new problems occurring has been substantially reduced, so much so, that within one week the D-Tree was installed on four different and relatively new environments with only one problem. That problem arose on the very first command of the installation and was reported with the diagnostic:

make: not found¹⁵

As part of this conclusion we must list some of the short-comings and discuss the future.

The approach (and its supporting tools) is primarily the result of one person’s research and is a UNIX prototype. As such, future or wider use may raise problems that have not been identified. The user community is still small and most of its applications are well controlled and usually performed by people versed in its use or with access to the creator. The following is a list of known problems and limitations that are partially a result of these factors.

- The 7001 version of the system is constantly changing and evolving, and some of these changes require modification to the existing pmak scripts which can cause users irritation at the quarterly upgrades.
- The approach seems overly complicated in some respects, since it reflects the complexity of the primary objective (to install over 1000 programs and data files on any UNIX system). If applied to smaller projects this complexity may raise more problems than it solves.
- The “-x” flag concept is effective only if applied to a large number of programs¹⁶.
- The syntactic inconsistencies and obtuseness of the pmak script generators’ inputs will have to be rectified if they are to be used by a larger community.
- The reorganisation of imported source to permit the effective use of pmak, sometimes complicates the incorporation of subsequent updates from the original distributor.
- There is a loss of flexibility caused by the source reorganisation, the naming conventions and the narrow range of *pmcmd*(1) provided constructions.
- The approach can give users a false sense of security, thus they fail to adequately check the results of a large pmak invocation.
- The addition of a new suffix to *pmcmd*(1) is non-trivial but fortunately occurs infrequently. *pmcmd*(1) should probably be converted into a *pmproto*(1) script. The only impediment is the differentiation between source and generated files when not distinguishable by suffix alone (e.g., *bill.y* vs. *bill.c*) and the resolution of conflicts (e.g., *fred.sh* vs. *fred.c*).
- The *instal*(1) supplementary flags file *Instflags.L* appears to have become an unnecessary luxury, and is currently under investigation.

15. The site in question uses a non-native make to avoid confusion and had removed the standard “/bin/make”.

16. Many of the standard tools UNIX tools, when invoked with a “-x” flag respond:

-x: not found

or perform some unknown processing, silently ignoring the specification of an unsupported flag. Unfortunately the increasing use of *getopts*(3) seems to ensure that the number of programs that mismanage arguments will rise.

Despite these problems, the policies and tools described in this paper seem to work and we appear to have resolved most of the problems identified in section 2. Furthermore, since these policies are easier to use than to ignore, and obviously beneficial, there has been little resistance from programming staff.

Finally, we must consider the future directions of this work. The frequency of changes to the overall strategy, policies and major tools seems to be decreasing.¹⁷ Most of the recent changes have been cosmetic or minor optimisations. It is now our opinion, that whilst the current approach is a huge improvement over those previously employed, further development would be unprofitable without advances in the following areas:

- the semantics, generation, expression, validation, control and algebras of dependencies and their extension to deal with version and environmental settings.
- an evaluation of *make(1)*'s suitability as a back-end and its simplification, enhancement and/or replacement to resolve the problems.

Currently there are a number of organisations working in the first area, using a variety of approaches. Unfortunately their objectives are widely diverging and dependent on many factors (e.g., managerial style) and there seems to be little progress or agreement. It is our belief that this is largely due to the fact that the problem is not going to be solved without experimentation and evolution, and that this cannot be done without the appropriate tools.

17. Please excuse our hysterical co-workers.

Vnodes: An Architecture for Multiple File System Types in Sun UNIX

S.R. Kleiman

Sun Microsystems
sun!srk

1. Introduction

This paper describes an architecture for accommodating multiple file system implementations within the Sun UNIX[†] kernel. The file system implementations can encompass local, remote, or even non-UNIX file systems. These file systems can be "plugged" into the kernel through a well defined interface, much the same way as UNIX device drivers are currently added to the kernel.

2. Design Goals

- Split the file system implementation independent and the file system implementation dependent functionality of the kernel and provide a well defined interface between the two parts.
- The interface must support (but not require) UNIX file system access semantics. In particular it must support local disk file systems such as the 4.2BSD file system^[1], stateless remote file systems such as Sun's NFS^[2], statefull remote file systems such as AT&T's RFS, or non-UNIX file systems such as the MSDOS file system^[3].
- The interface must be usable by the server side of a remote file system to satisfy client requests.
- All file system operations should be atomic. In other words, the set of interface operations should be at a high enough level so that there is no need for locking (hard locking, not user advisory locking) across several operations. Locking, if required, should be left up to the file system implementation dependent layer. For example, if a relatively slow computer running a remote file system requires a supercomputer server to lock a file while it does several operations, the users of the supercomputer would be noticeably affected. It is much better to give the file system dependent code full information about what operation is being done and let it decide what locking is necessary and practical.

3. Implementation goals and techniques

These implementation goals were necessary in order to make future implementation easier as the kernel evolved.

- There should be little or no performance degradation.
- The file system independent layer should not force static table sizes. Most of the new file system types use a dynamic storage allocator to create and destroy objects.
- Different file system implementations should not be forced to use centralized resources (e.g inode table, mount table or buffer cache). However, sharing should be allowed.
- The interface should be reentrant. In other words, there should be no implicit references to global data (e.g. `u.u_base`) or any global side effect information passed between operations (e.g. `u.u_dent`). This has the added benefit of cutting down the size of the per user global data area (`u` area). In addition, all the interface operations return error codes as the return value. Overloaded return codes and `u.u_error` should not be used.
- The changes to the kernel should be implemented by an "object oriented" programming approach. Data structures representing objects contain a pointer to a vector of generic operations on the object. Implementations of the object fill in the vector as appropriate. The complete interface to the object is

[†] UNIX is a trademark of AT&T

specified by its data structure and its generic operations. The object data structures also contain a pointer to implementation specific data. This allows implementation specific information to be hidden from the interface.

- Each interface operation is done on behalf of the current process. It is permissible for any interface operation to put the current process to sleep in the course of performing its function.

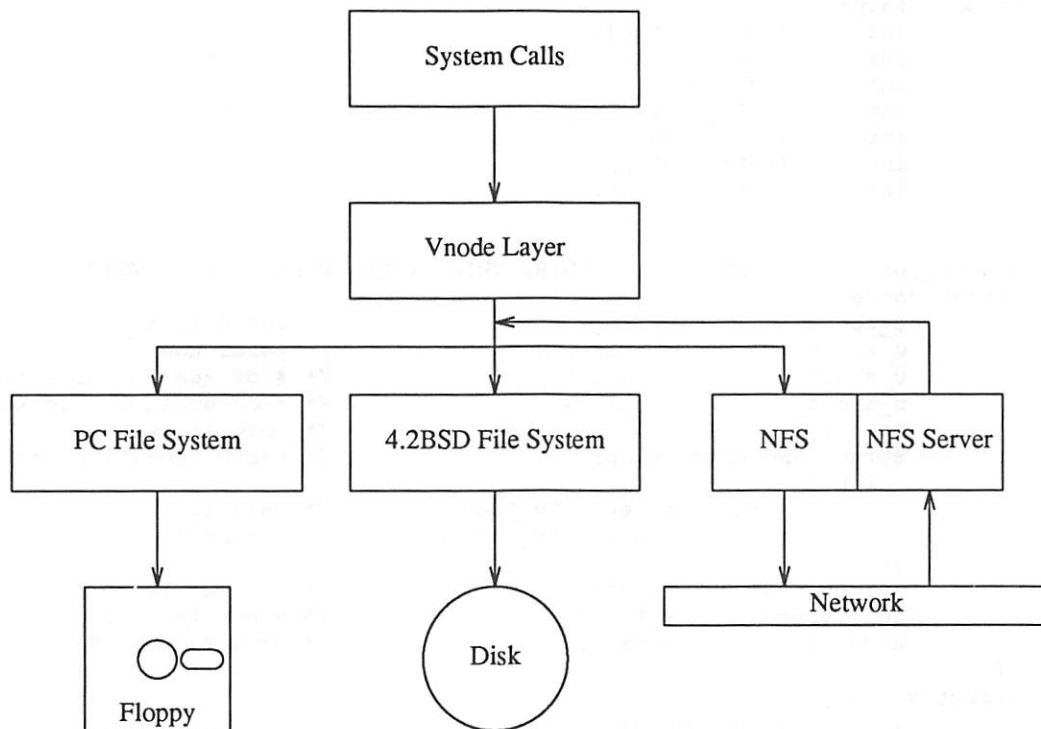


Figure 1. Vnode architecture block diagram

4. Operation

The file system dependent/independent split was done just above the UNIX kernel inode layer. This was an obvious choice, as the inode was the main object for file manipulation in the kernel. A block diagram of the architecture is shown in Figure 1. The file system independent inode was renamed *vnode* (virtual node). All file manipulation is done with a *vnode* object. Similarly, file systems are manipulated through an object called a *vfs* (virtual file system). The *vfs* is the analog to the old mount table entry. The file system independent layer is generally referred to as the *vnode layer*. The file system implementation dependent layer is called by the file system type it implements (e.g. 4.2BSD file system, NFS file system). Figure 2 shows the definition of the *vnode* and *vfs* objects.

4.1 Vfs's

Each mounted *vfs* is linked into a list of mounted file systems. The first file system on the list is always the root. The private data pointer (*vfs_data*) in the *vfs* points to file system dependent data. In the 4.2BSD file system, *vfs_data* points to a mount table entry. The public data in the *vfs* structure contains data used by the *vnode layer* or data about the mounted file system that does not change.

Since different file system implementations require different mount data, the *mount(2)* system call was changed. The arguments to *mount(2)* now specify the file system type, the directory which is the mount point, generic flags (e.g. read only), and a pointer to file system type specific data. When a mount system

```

struct vfs {
    struct vfs      *vfs_next;           /* next vfs in list */
    struct vfsops   *vfs_op;            /* operations on vfs */
    struct vnode    *vfs_vnodecovered;  /* vnode we cover */
    int             vfs_flag;           /* flags */
    int             vfs_bsize;          /* native block size */
    caddr_t         vfs_data;           /* private data */
};

struct vfsops {
    int             (*vfs_mount) ();
    int             (*vfs_unmount) ();
    int             (*vfs_root) ();
    int             (*vfs_statfs) ();
    int             (*vfs_sync) ();
    int             (*vfs_fid) ();
    int             (*vfs_vget) ();
};

enum vtype { VNON, VREG, VDIR, VBLK, VCHR, VLNK, VSOCK, VBAD };

struct vnode {
    u_short         v_flag;             /* vnode flags */
    u_short         v_count;            /* reference count */
    u_short         v_shlockc;          /* # of shared locks */
    u_short         v_exlockc;          /* # of exclusive locks */
    struct vfs      *v_vfsmountedhere; /* covering vfs */
    struct vnodeops *v_op;              /* vnode operations */
    union {
        struct socket *v_Socket;       /* unix ipc */
        struct stdata *v_Stream;       /* stream */
    };
    struct vfs      *v_vfsp;            /* vfs we are in */
    enum vtype      v_type;             /* vnode type */
    caddr_t         v_data;             /* private data */
};

struct vnodeops {
    int             (*vn_open) ();
    int             (*vn_close) ();
    int             (*vn_rdwr) ();
    int             (*vn_ioctl) ();
    int             (*vn_select) ();
    int             (*vn_getattr) ();
    int             (*vn_setattr) ();
    int             (*vn_access) ();
    int             (*vn_lookup) ();
    int             (*vn_create) ();
    int             (*vn_remove) ();
    int             (*vn_link) ();
    int             (*vn_rename) ();
    int             (*vn_mkdir) ();
    int             (*vn_rmdir) ();
    int             (*vn_readdir) ();
    int             (*vn_symlink) ();
    int             (*vn_readlink) ();
    int             (*vn_fsync) ();
    int             (*vn_inactive) ();
    int             (*vn_bmap) ();
    int             (*vn_strategy) ();
    int             (*vn_bread) ();
    int             (*vn_brelse) ();
};

```

Figure 2. Vfs and vnode objects

call is performed, the vnode for the mount point is looked up (see below) and the `vfs_mount` operation for the file system type is called. If this succeeds, the file system is linked into the list of mounted file systems, and the `vfs_vnodecovered` field is set to point to the vnode for the mount point. This field is null in the root vfs. The root vfs is always first in the list of mounted file systems.

Once mounted, file systems are named by the path name of their mount points. Special device name are no longer used because remote file systems do not necessarily have a unique local device associated with them. `Umount(2)` was changed to `umount(2)` which takes a path name for a file system mount point instead of a device.

The root vnode for a mounted file system is obtained by the `vfs_root` operation, as opposed to always referencing the root vnode in the vfs structure. This allows the root vnode to be deallocated if the file system is not being referenced. For example, remote mount points can exist in "embryonic" form, which contains just enough information to actually contact the server and complete the remote mount when the file system is referenced. These mount points can exist with minimal allocated resources when they are not being used.

4.2 Vnodes

The public data fields in each vnode either contain data that is manipulated only by the vfs layer or data about the file that does not change over the life of the file, such as the file type (`v_type`). Each vnode contains a reference count (`v_count`) which is maintained by the generic vnode macros `VN_HOLD` and `VN_RELE`. The vnode layer and file systems call these macros when vnode pointers are copied or destroyed. When the last reference to a vnode is destroyed, the `vn_inactive` operation is called to tell the vnode's file system that there are no more references. The file system may then destroy the vnode or cache it for later use. The `v_vfsp` field in the vnode points to the vfs for the file system to which the vnode belongs. If a vnode is a mount point, the `v_vfsmountedhere` field points to the vfs for another file system. The private data pointer (`v_data`) in the vnode points to data that is dependent on the file system. In the 4.2BSD file system `v_data` points to an in core inode table entry.

Vnodes are not locked by the vnode layer. All hard locking (i.e. not user advisory locks) is done within the file system dependent layer. Locking could have been done in the vnode layer for synchronization purposes without violating the design goal; however, it was found to be not necessary.

4.3 An example

Figure 3 shows an example vnode and vfs object interconnection. In figure 3, `vnodel` is a file or directory in a 4.2BSD type file system. As such, it's private data pointer points to an inode in the 4.2BSD file system's inode table. `vnodel` belongs to `vfsl`, which is the root vfs, since it is the first on the vfs list (`rootvfs`). `vfsl`'s private data pointer points to a mount table entry in the 4.2BSD file system's mount table. `Vnode2` is a directory in `vfsl`, which is the mount point for `vfs2`. `Vfs2` is an NFS file system, which contains `vnodel3`.

4.4 Path name traversal

Path name traversal is done by the `lookupn` routine (lookup path name), which takes a path name in a path name buffer and returns a pointer to the vnode which the path represents. This takes the place of the old `namei` routine.

If the path name begins with a "/", Path name traversal starts at the vnode pointed to by either `u.u_rdir` or the root. Otherwise it starts at the vnode pointed to by `u.u_cdir` (the current directory). `Lookupn` traverses the path one component at a time using the `vn_lookup` vnode operation. `Vn_lookup` takes a directory vnode and a component as arguments and returns a vnode representing that component. If a directory vnode has `v_vfsmountedhere` set, then it is a mount point. When a mount point is encountered going down the file system tree, `lookupn` follows the vnode's `v_vfsmountedhere` pointer to the mounted file system and calls the `vfs_root` operation to obtain the root vnode for the file system. Path name traversal then continues from this point. If a root vnode is encountered (VROOT flag in `v_flag` set) when following "!", `lookupn` follows the `vfs_vnodecovered` pointer in the vnode's associated vfs to obtain the covered vnode. If a symbolic link is encountered `lookupn` calls the

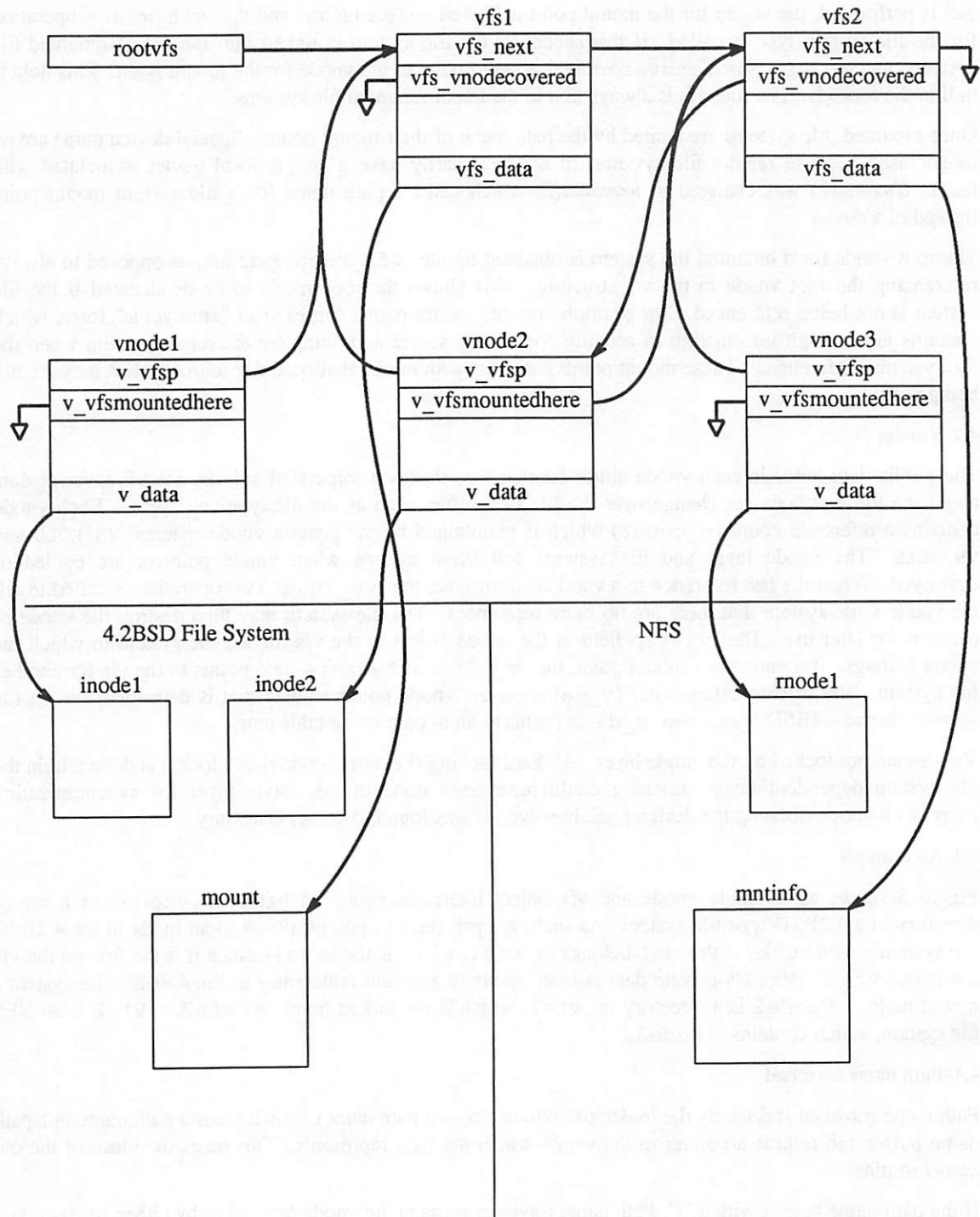


Figure 3. Example vnode layer object interconnection

`vn_readlink` vnode operation to obtain the symbolic link. If the symbolic link begins with a "/", the path name traversal is restarted from the root (or `u.u_rdir`); otherwise the traversal continues from the last directory. The caller of `lookuppn` specifies whether the last component of the path name is to be followed if it is a symbolic link. This process continues until the path name is exhausted or an error occurs. When `lookuppn` completes, a vnode representing the desired file is returned.

4.5 Remote file systems

The path name traversal scheme implies that files on remote file systems appear as files within the normal UNIX file name space. Remote files are not named by any special constructs that current programs don't understand^[4]. The path name traversal process handles all indirection through mount points. This means that in a remote file system implementation, the client maintains its own mount points. If the client mounts another file system on a remote directory, the remote file system will never see references below the new mount point. Also, the remote file system will not see any ".." references at the root of the remote file system. For example, if the client has mounted a server's "/usr" on his "/usr" and a local file system on "/usr/local" then the path "/usr/local/bin" will access the local root, the remote "/usr" and the local "/usr/local" without the remote file system having any knowledge of the "/usr/local" mount point. Similarly, the path "/usr/.." will access the local root, the remote "/usr" and the local root, without the remote file system (or the server) seeing the ".." out of "/usr".

4.6 New system calls

Three new system calls were added in order to make the normal application interface file system implementation independent. The `getdirentries(2)` system call was added to read directories in a manner which is independent of the on disk directory format. `Getdirentries` reads directory entries from an open directory file descriptor into a user buffer, in file system independent format. As many directory entries as can fit in the buffer are read. The file pointer is changed so that it points at directory entry boundaries after each call to `getdirentries`. The `statfs(2)` and `fstatfs(2)` system calls were added to get general file system statistics (e.g. space left). `Statfs` and `fstatfs` take a path name or a file descriptor, respectively, for a file within a particular file system, and return a `statfs` structure (see below).

4.7 Devices

The device interfaces, `bdevsw` and `cdevsw`, are hidden from the vnode layer, so that devices are only manipulated through the vnode interface. A special device file system implementation, which is never mounted, is provided to facilitate this. Thus, file systems which have a notion of associating a name within the file system with a local device may redirect vnodes to the special device file system.

4.8 The buffer cache

The buffer cache routines have been modified to act either as a physical buffer cache or a logical buffer cache. A local file system typically uses the buffer cache as a cache of physical disk blocks. Other file system types may use the buffer cache as a cache of logical file blocks. Unique blocks are identified by the pair (vnode-pointer, block-number). The vnode pointer points to a device vnode when a cached block is a copy of a physical device block, or it points to a file vnode when the block is a copy of a logical file block.

5. VFS operations

In the following descriptions of the vfs operations the `vfsp` argument is a pointer to the vfs that the operation is being applied to.

- | | |
|--|--|
| <code>vfs_mount(vfsp, pathp, datap)</code> | Mount <i>vfsp</i> (i.e. read the superblock etc.). <i>Pathp</i> points to the path name to be mounted (for recording purposes), and <i>datap</i> points to file system dependent data. |
| <code>vfs_unmount(vfsp)</code> | Unmount <i>vfsp</i> (a.e. sync the superblock). |
| <code>vfs_root(vfsp, vpp)</code> | Return the root vnode for this file system. <i>Vpp</i> points to a pointer to a vnode for the results. |

`vfs_statfs(vfsp,sbp)`

Return file system information. *Sbp* points to a statfs structure for the results.

```
struct statfs {
    long f_type;           /* type of info */
    long f_bsize;          /* block size */
    long f_blocks;         /* total blocks */
    long f_bfree;          /* free blocks */
    long f_bavail;         /* non-su blocks */
    long f_files;          /* total # of nodes */
    long f_ffree;          /* free nodes in fs */
    fsid_t f_fsid;         /* file system id */
    long f_spare[7];       /* spare for later */
};
```

`vfs_sync(vfsp)`

Write out all cached information for *vfsp*. Note that this is not necessarily done synchronously. When the operation returns all data has not necessarily been written out, however it has been scheduled.

`vfs_fid(vfsp,vp,fidpp)`

Get a unique file identifier for *vp* which represents a file within this file system. *Fidpp* points to a pointer to a fid structure for the results.

```
struct fid {
    u_short fid_len;       /* length of data */
    char fid_data[1];      /* variable size */
};
```

`vfs_vget(vfsp,vpp,fidp)`

Turn unique file identifier *fidp* into a vnode representing the file associated with the file identifier. *vpp* points to a pointer to a vnode for the result.

6. Vnode operations

In the following descriptions of the vnode operations, the *vp* argument is a pointer to the vnode to which the operation is being applied; the *c* argument is a pointer to a credentials structure which contains the user credentials (e.g. uid) to use for the operation; and the *nm* argument is a pointer to a character string containing a name.

`vn_open(vpp,f,c)`

Perform any open protocol on a vnode pointed to by *vpp* (e.g. devices). If the open is a "clone" open the operation may return a new vnode. *F* is the open flags.

`vn_close(vp,f,c)`

Perform any close protocol on a vnode (e.g. devices). Called on the closing of the last reference to the vnode from the file table, if vnode is a device. Called on the last user close of a file descriptor, otherwise. *F* is the open flags.

`vn_rdwr(vp,uiop,rw,f,c)`

Read or write vnode. Reads or writes a number of bytes at a specified offset in the file. *Uiop* points to a *uio* structure which supplies the I/O arguments. *Rw* specifies the I/O direction. *F* is the I/O flags, which may specify that the I/O is to be done synchronously (i.e. don't return until all the volatile data is on disk) and/or in a unit (i.e. lock the file to write a large unit).

`vn_ioctl(vp,com,d,f,c)`

Perform an *ioctl* on vnode *vp*. *Com* is the command, *d* is the pointer to the data, and *f* is the open flags.

`vn_select(vp,w,c)`

Perform a *select* on *vp*. *W* specifies the I/O direction.

`vn_getattr(vp,va,c)`

Get attributes for *vp*. *Va* points to a *vattr* structure.

```

struct vattr {
    enum vtype      va_type;      /* vnode type */
    u_short         va_mode;      /* acc mode */
    short           va_uid;       /* owner uid */
    short           va_gid;       /* owner gid */
    long            va_fsid;      /* fs id */
    long            va_nodeid;    /* node # */
    short           va_nlink;     /* # links */
    u_long          va_size;      /* file size */
    long            va_blocksize; /* block size */
    struct timeval  va_atime;     /* last acc */
    struct timeval  va_mtime;     /* last mod */
    struct timeval  va_ctime;     /* last chg */
    dev_t           va_rdev;      /* dev */
    long            va_blocks;    /* space used */
};

```

This must map file system dependent attributes to UNIX file attributes.

- `vn_setattr(vp,va,c)` Set attributes for *vp*. *Va* points to a *vattr* structure, but only mode, uid, gid, file size, and times may be set. This must map UNIX file attributes to file system dependent attributes.
- `vn_access(vp,m,c)` Check access permissions for *vp*. Returns error if access is denied. *M* is the mode to check for access (e.g. read, write, execute). This must map UNIX file protection information to file system dependent protection information.
- `vn_lookup(vp,nm,vpp,c)` Lookup a component name *nm* in directory *vp*. *Vpp* points to a pointer to a vnode for the results.
- `vn_create(vp,nm,va,e,m,vpp,c)` Create a new file *nm* in directory *vp*. *Va* points to an *vattr* structure containing the attributes of the new file. *E* is the exclusive/non-exclusive create flag. *M* is the open mode. *vpp* points to a pointer to a vnode for the results.
- `vn_remove(vp,nm,c)` Remove a file *nm* in directory *vp*.
- `vn_link(vp,tdvp,tnm,c)` Link the vnode *vp* to the target name *tnm* in the target directory *tdvp*.
- `vn_rename(vp,nm,tdvp,tnm,c)` Rename the file *nm* in directory *vp* to *tnm* in target directory *tdvp*. The node can't be lost if the system crashes in the middle of the operation.
- `vn_mkdir(vp,nm,va,vpp,c)` Create directory *nm* in directory *vp*. *Va* points to an *vattr* structure containing the attributes of the new directory and *vpp* points to a pointer to a vnode for the results.
- `vn_rmdir(vp,nm,c)` Remove the directory *nm* from directory *vp*.
- `vn_readdir(vp,uiop,c)` Read entries from directory *vp*. *Uiop* points to a *uio* structure which supplies the I/O arguments. The *uio* offset is set to a file system dependent number which represents the logical offset in the directory when the reading is done. This is necessary because the number of bytes returned by *vn_readdir* is not necessarily the number of bytes in the equivalent part of the on disk directory.
- `vn_symlink(vp,lrm,va,tnm,c)` Symbolically link the path pointed to by *tnm* to the name *lrm* in directory *vp*.
- `vn_readlink(vp,uiop,c)` Read symbolic link *vp*. *Uiop* points to a *uio* structure which supplies the I/O arguments.
- `vn_fsync(vp,c)` Write out all cached information for file *vp*. The operation is synchronous and does not return until the I/O is complete.

<code>vn_inactive(vp,c)</code>	The <i>vp</i> is no longer referenced by the vnode layer. It may now be deallocated.
<code>vn_bmap(vp,bn,vpp,bnp)</code>	Map logical block number <i>bn</i> in file <i>vp</i> to physical block number and physical device. <i>Bnp</i> is a pointer to a block number for the physical block and <i>vpp</i> is a pointer to a vnode pointer for the physical device. Note that the returned vnode is not necessarily a physical device. This is used by the paging system to premap files before they are paged. In the NFS this is a null mapping.
<code>vn_strategy(bp)</code>	Block oriented interface to read or write a logical block from a file into or out of a buffer. <i>Bp</i> is a pointer to a buffer header which contains a pointer to the vnode to be operated on. Does not copy through the buffer cache if the file system uses it. This is used by the buffer cache routines and the paging system to read blocks into memory.
<code>vn_bread(vp,bn,bpp)</code>	Read a logical block <i>bn</i> from a file <i>vp</i> and return a pointer to a buffer header in <i>bpp</i> which contains a pointer to the data. This does not necessarily imply the use of the buffer cache. This operation is useful avoid extra data copying on the server side of a remote file system.
<code>vn_brelse(vp,bp)</code>	The buffer returned by <i>vn_bread</i> can be released.

6.1 Kernel interfaces

A veneer layer is provided over the generic vnode interface to make it easier for kernel subsystems to manipulate files:

<code>vn_open</code>	Perform permission checks and then open a vnode given by a path name.
<code>vn_close</code>	Close a vnode.
<code>vn_rdwr</code>	Build a <i>uio</i> structure and read or write a vnode.
<code>vn_create</code>	Perform permission checks and then create a vnode given by a path name.
<code>vn_remove</code>	Remove a node given by a path name.
<code>vn_link</code>	Link a node given by a source path name to a target given by a target path name.
<code>vn_rename</code>	Rename a node given by a source path name to a target given by a target path name.
<code>VN_HOLD</code>	Increment the vnode reference count.
<code>VN_RELE</code>	Decrement the vnode reference count and call <i>vn_inactive</i> if this is the last reference.

Many system calls which take names do a *lookuppn* to resolve the name to a vnode then call the appropriate veneer routine to do the operation. System calls which work off file descriptors pull the vnode pointer out of the file table and call the appropriate veneer routine.

7. Current status

The current interface has been in operation since the summer of 1984, and is a released Sun product. In general, the system performance degradation was nil to 2% depending on the benchmark. To date the 4.2BSD file system, the Sun Network File System, and an MSDOS floppy disk file system have been implemented under the interface, with other file system types to follow. In addition, a prototype "/proc" file system^[5] has been implemented. It is also possible to configure out all the disk based file systems and run with just the NFS. Throughout this time the interface definition has been stable, with minor additions, even though several radically different file system types were implemented. Vnodes has been proven to provide a clean, well defined interface to different file system implementations.

Sun is currently discussing with AT&T and Berkeley the merging of this interface with AT&T's File System Switch technology. The goal is to produce a standard UNIX file system interface. Some of the

current issues are:

- Allow multiple component lookup in `vn_lookup`. This would require file systems that implemented this to know about mount points.
- Cleaner replacements for `vn_bmap`, `vn_strategy`, `vn_bread`, and `vn_brelse`.
- Symlink handling in the file system independent layer.
- Eliminate redundant lookups.

8. Acknowledgements

Bill Joy is the designer of the architecture, and provided much help in its implementation. Dan Walsh modified *bio* and implemented parts of the device interface as well as parts of the 4.2BSD file system port. Russel Sandberg was the primary NFS implementor. He also built the prototype `/proc` file system and improved the device interface. Bill Shannon, Tom Lyon and Bob Lyon were invaluable in reviewing and evolving the interface.

REFERENCES

1. M.K. McKusick, W. Joy, S. Leffler, R. Fabry, "A Fast File System for UNIX, ACM TOCS, 2, 3, August 1984, pp 181-197.
2. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, B. Lyon, "Design and Implementation of the Sun Network Filesystem", USENIX Summer 1985, pp 119-130.
3. IBM, "DOS Operating System Version 2.0", January 1983.
4. R. Pike, P. Weinberger, "The Hideous Name" USENIX Summer 1985, pp 563-568.
5. T.J. Killian, "Processes as Files", USENIX Summer 1985, pp 203-207.

RFS Architectural Overview

*Andrew P. Rifkin
Michael P. Forbes
Richard L. Hamilton
Michael Sabrio
Suryakanta Shah
Kang Yueh*

AT&T
190 River Road
Summit, NJ 07901

ABSTRACT

Remote File Sharing (RFS) is one of the networking based features offered in AT&T's UNIX* System V Release 3.0 (SVR3). RFS adds a new dimension to the user computing environment by providing transparent access to remote files. RFS allows access of all file types, including special devices and named pipes, in addition to allowing file and record locking on remote files. Careful attention to preserving the UNIX file system semantics ensures that existing binary applications can make use of network resources without failure.

By extending the notion of the UNIX mount, RFS allows a subtree of a server machine to be logically added to the local file tree of a client machine. A message protocol based on the UNIX system call interface is used to communicate resource requests between the machines. The client and server machines employ a reliable virtual circuit style transport to transfer these messages. By adhering to a standard transport service interface accessed via the STREAMS^[1] mechanism, RFS can operate over a wide variety of commercially available protocols without modification.

1. Introduction

The Remote File Sharing (RFS) feature of UNIX System V Release 3.0 (SVR3) is AT&T's offering to the distributed file system market. RFS provides the user with transparent access to remote files. Unlike other distributed file systems RFS preserves full UNIX file system semantics and allows access to all types of files including special devices and named pipes. In addition RFS extends file and record locking to remote files.

This paper describes the set of goals on which RFS is based, the architecture used to achieve these goals, and details of the RFS implementation.

2. RFS Goals

The main goal of RFS is to provide users and applications a means of accessing remote files. In the course of attaining this goal several subgoals were defined.

* UNIX is a trademark of AT&T.

Transparent Access

The standard UNIX interface must be preserved. Access of a remote file must be the same as a local file. Accessing remote files must be independent of the file's physical location.

Semantics

The UNIX System semantics must be preserved. All file types including special devices and named pipes must be accessible through RFS. File and record locking on remote files must be supported.

Binary Compatibility

Existing applications must not require modification or recompilation to make use of network resources.

Network Independence

In light of the rapid pace at which network products are evolving, it was decided that RFS should be cleanly separated from the underlying network. RFS must operate, without modification, over a variety of networks that range from LAN's to large concatenated networks.

Portability

RFS code must be as machine independent as possible to ease porting to different hardware environments. To simplify the integration of RFS into various UNIX Systems the changes to the kernel were localized and kept to a minimum.

Performance

Considering that a large cost of any remote operation is the network overhead, the performance goal was to minimize network access.

3. RFS Architecture

The RFS architecture is based on a client/server model using a central name server and a remote mounting scheme for connection establishment. Once connected the machines communicate using a message protocol based on the UNIX system call interface.

The STREAMS mechanism in conjunction with the transport interface defined in System V is used to separate RFS from the underlying network, making RFS network independent.

By defining an RFS file system type, RFS is cleanly integrated into the UNIX kernel using the File System Switch (FSS) mechanism in SVR3.

To ensure security the normal UNIX file protection is extended to remote files and a mechanism is provided to map user id's.

3.1 Client/Server

A file sharing relationship consists of two machines, a client machine and a server machine. The file physically resides on the server machine, while the client machine remotely accesses the file. The client accesses the file by sending a request message to the server machine. The server provides the resource in the form of a response message to the client.

Any machine may be a client, server or both.

3.2 Connection Establishment

The RFS connection establishment involves locating and identifying a remote resource followed by remotely mounting it. The location and identification of resources is done using the RFS name server. The remote mount adds the remote file system to the local file tree. The remote mount model will be described followed by a discussion of the RFS name server.

Remote Mount

The remote mount model provides the same "tree building" approach used in the UNIX System. A client machine can add (mount) a remote file system from a server machine onto its local file tree. To support this a two step process is required. First, the server must (advertise) make a subtree of its local file tree available. Second, the client must add (mount) this subtree onto its local file tree (Figure 1).

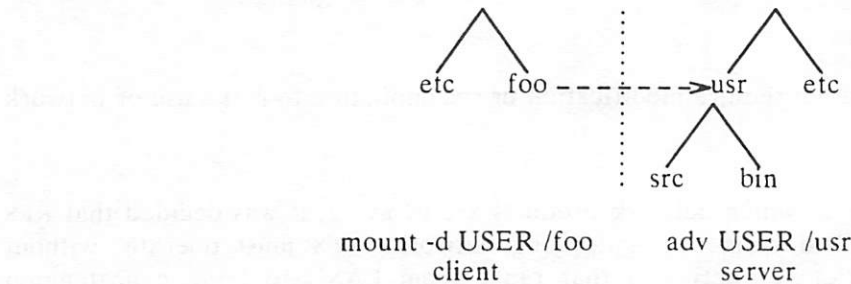


Figure 1. Remote Mount Model

When a subtree is advertised a symbolic name is assigned to it by the server. In Figure 1, the /usr subtree is assigned the name **USER**. A client machine now uses this symbolic name to mount this file system onto its local file tree.

Name Server

The RFS network should be viewed as a network of file systems rather than a network of machines sharing file systems. Therefore, it is necessary to logically separate a resource from its location. The RFS name server does just that.

The RFS name server maps resource names, which represent file trees that are available to share, into information about that resource. It allows machines to register the name of a resource, and it allows other machines to make queries about what resources are available. To accomplish this the name server maintains a centralized data base with a reliable recovery mechanism to avoid a single point of failure. This data base contains all currently advertised resources, mapping the symbolic name to the network location. The name server enforces uniqueness of symbolic names within a domain (see below) ensuring a consistent network view.

When a resource is advertised, the name server checks the symbolic name for uniqueness and if unique registers the resource in the data base. When the resource is mounted the name server converts the user specified symbolic name to the network location.

Domains

As a network grows, resource management becomes increasingly difficult. To alleviate this problem a domain based naming scheme is used. This concept allows machines to be logically grouped into a smaller, separate name space called a domain. For instance, all machines belonging to a single department in a large corporate structure may be partitioned into a single domain, carving the large corporate network into smaller more manageable pieces.

To reference a resource within the local domain specifying the symbolic name is sufficient. To reference a resource from another domain it is necessary to prefix the symbolic name with the name of the domain in which the resource resides (Figure 2).

3.3 RFS Message Protocol

The RFS message protocol is based on the UNIX system calls, which are well defined^[2] and accepted. This protocol is used to communicate remote resource requests between client and server machines.

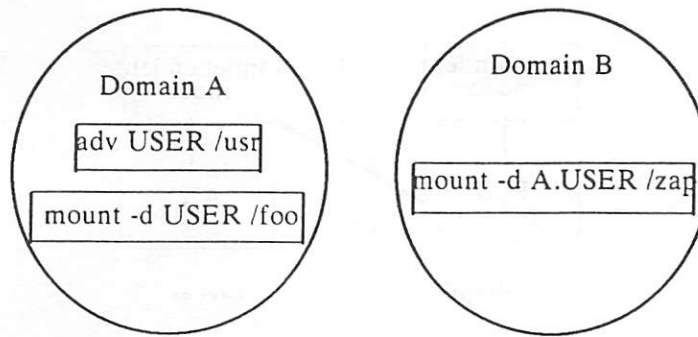


Figure 2. Domain Naming Scheme

For each system call there exists a request and response message. The request message formats all pertinent information necessary to execute the system call, while the response message formats all possible results. The following brief description demonstrates the use of this protocol.

A client process, in the course of executing a system call encounters a remote resource. Execution of the system call is suspended, the client's environment data is copied into a request message, and the message is sent to the server machine. On the server machine, a server process services the request by recreating the client's environment based on the contents of the request message and executes the specified system call. The results of the system call are copied into a response message and sent to the client machine. Therefore, a remote system call requires only two messages a request message and a response message, thus minimizing network access.

3.4 RFS File System Type

The File System Switch (FSS) mechanism of UNIX SVR3 allows the same UNIX operating system to simultaneously support different file system implementations. Based on Peter Weinberger's (AT&T Bell Laboratories UNIX Research Laboratory) inode level switch, the FSS mechanism preserves system call compatibility while isolating different file system implementations from one another.

FSS separates the generic file system information from the file system specific information, and defines a common interface between the kernel and the underlying file systems. This is done by dividing the inode into two parts, a file system independent portion (generic information) and a file system dependent portion. FSS intervenes between the kernel and the file system by directing inode operations from the kernel to the file system specified by the "type" of the dependent inode.

RFS makes use of this feature by defining an RFS file system type. This file system type defines RFS type dependent inodes, which contain a communication pointer across the network to the "real" inode on the server machine (Figure 3). Now, inode operations resulting from a file descriptor based system call are directed to the RFS module via the FSS. The RFS module in turn, uses the communication pointer in the RFS dependent inode to direct the request message.

3.5 Network Independence

By separating RFS from the underlying transport service, RFS is able to run over a variety of protocols and networks without modification. To accomplish this two problems had to be solved. First, it was necessary to choose what style of transport service RFS required. Second, a standard interface between RFS and the transport provider was needed.

Since RFS must work over a large concatenated network, and since the overhead of retransmission and error detection is high for datagram service over large networks, reliable

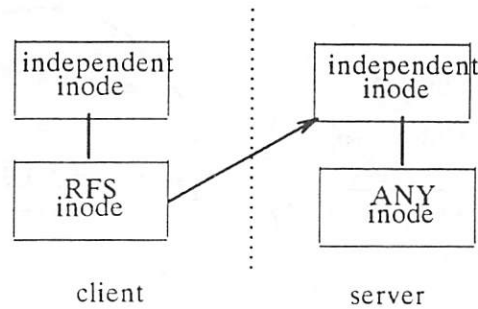


Figure 3. RFS File System Type

virtual circuit service was chosen. To compensate for virtual circuit setup costs, a single virtual circuit is maintained between a client and server machine. This circuit is established during the first remote mount, and all subsequent mounts are multiplexed over this circuit. The circuit is held open for the duration of the mounts.

The second problem was solved using the transport interface (based on ISO Transport Service Definition) defined within the System V networking framework^[3] and the STREAMS mechanism. By adhering to the transport interface, a connection between RFS and a transport provider (also adhering to the transport interface) is provided via the STREAMS mechanism. In addition, by allowing RFS to communicate over multiple STREAMS, RFS is able to make use of a variety of transport providers simultaneously (Figure 4).

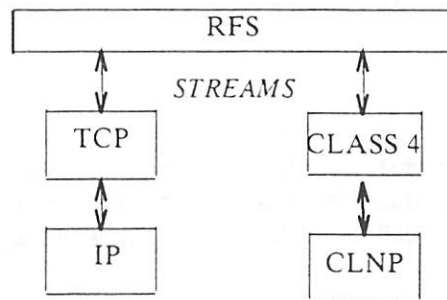


Figure 4. STREAMS Based Communication

3.6 Security

One of the problems in allowing one machine to transparently access files on another is the need to authorize the access. Two levels of security must be considered, the machine level and the user level. At the machine level a means of restricting clients from mounting a particular resource is provided. When a server machine advertises a resource the server can specify a list of clients that are allowed to mount the resource, restricting all other clients. In addition, a server machine can be configured to require a password check at the time a virtual circuit is established to the client machine.

At the user level the local UNIX security scheme has been extended to the network environment. In the local case access permission is based on a user's user identification (uid) and group identification (gid) compared to the file access rights. To make this scheme work for the remote case two problems must be solved. First, over a large network, a common password file (/etc/passwd) among a group of machines can not be guaranteed. Second, a means of restricting remote user access must be possible.

Both these problems are solved using a uid/gid mapping scheme. This scheme allows a uid/gid from a client machine to be mapped to a different uid/gid on the server machine. In the case of different password files, the uid/gid of a user on a client machine can be mapped

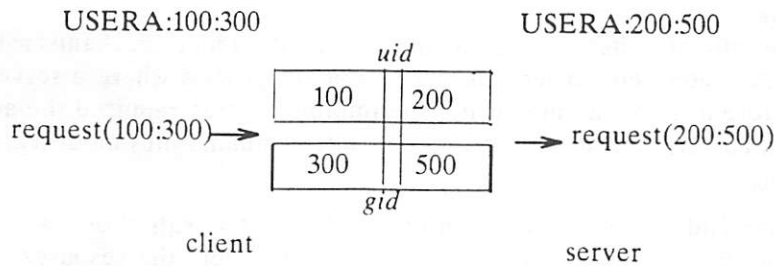


Figure 5. uid/gid mapping

to the uid/gid of that same user on the server machine (Figure 5). Users can similarly be restricted on the server machine by mapping their uid/gid to an impotent value. Figure 6 shows how super user on a client machine is restricted from having super user privileges on the server machine.

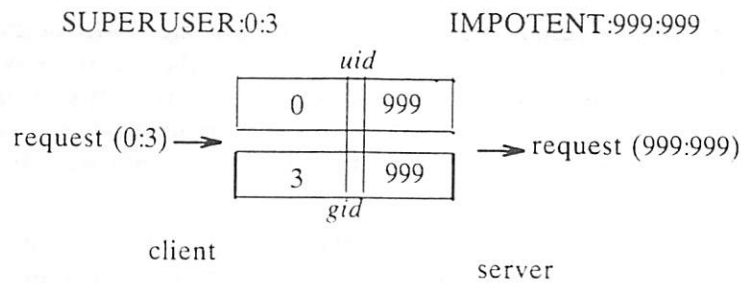


Figure 6. Super User Restriction

4. RFS Implementation

To minimize kernel change and to ease future porting, RFS was implemented as a separate module with a well defined interface into the kernel. The RFS module consists of approximately twenty source files and six header files.

The implementation was done in functional pieces. The *name server* was implemented separately, entirely at the user level. The *remote mount* code which establishes the data structures required to connect the client and server machines was separated from the remote access code. The remote access code was separated into *client* and *server* routines. This code makes up the bulk of the system. Finally, since RFS retains client state information on the server machine, a *recovery* mechanism was implemented to resynchronize state information, in the event of a machine crash. The implementation details of these components will be discussed below.

Name Server

The RFS name server is a user level daemon that is initiated on each machine when RFS is started. Processes that want to communicate with the RFS name server open a stream pipe device that is associated with the name server and use that stream pipe to send their requests. Communication between name servers on different machines uses the standard transport interface to provide protocol independence.

The RFS name server is modeled as a transaction handler; it receives a request, performs an operation, generates a reply, and sends the reply to the originator of the request. It also acts as an agent of the requesting process if a request needs to go to a remote name server. A request first goes to the local name server daemon, which services the request if it can and otherwise forwards it to the appropriate name server. A non-recursive method is used for finding a name server that has the required information to avoid looping.

Remote Mount

The remote mount scheme extends the standard mount mechanism to include remote resources. As described earlier this is a two phase process where a server must advertise a resource before a client can mount it. The implementation required the addition of two new system calls, *advfs()* and *rmount()* and a new *adv* command plus modifications to the existing *mount* command.

The *adv* command in conjunction with the *advfs()* system call allows a server to advertise a subtree of its local file tree. The *adv* command registers the resource in the name server data base. The command then calls *advfs()*. The *advfs()* system call stores the symbolic name, and a pathname for the subtree (which is resolved to an inode), in a kernel advertise table.

The *mount* command has been extended to mount remote resources. The new *-d* option now specifies that a remote resource is to be mounted. For example:

```
mount -d USER /foo
```

requests that the remote resource associated with the name **USER** be mounted on the local mount point **/foo**. The name server is used to convert the specified symbolic name to the network location. If a virtual circuit does not currently exist between the client and server, the *mount* command sets one up. This will be discussed in detail below. The *mount* command then uses the *rmount()* system call to establish the kernel data structures required for the remote mount.

The *rmount()* system call takes the symbolic name, local mount point (pathname), and virtual circuit pointer as arguments. A remote *rmount* request, which includes the symbolic name, is sent to the server via the virtual circuit indicated by the virtual circuit pointer. The server uses the advertise table and symbolic name to locate the inode of the desired resource. If this client is authorized for this resource, the server records the client access in a kernel server mount table. Each time a client mounts a particular resource an entry is placed in this table indicating the client's system identification and the mount table index that the client used to record this mount. This information is used to resolve pathnames which traverse back out of a remote resource by using a *".."* in the pathname. A response is then sent to the client containing a communication pointer to the inode of the advertised resource. Using this communication pointer the client creates an RFS inode. The RFS inode and the inode of the local mount point are then stored in the client's kernel mount table (Figure 7).

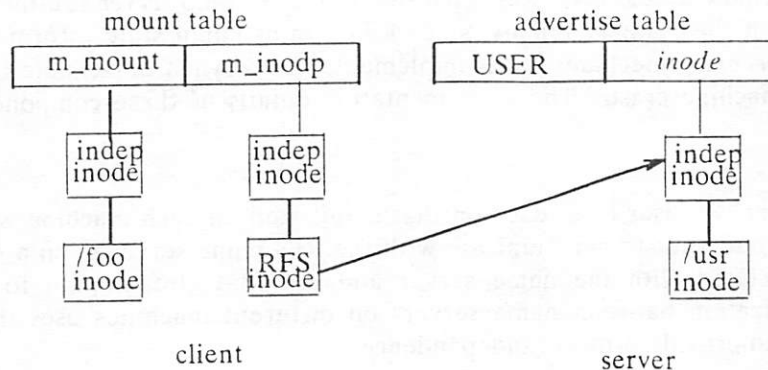


Figure 7. Remote Mount Data Structures

As mentioned before the *mount* command sets up a virtual circuit between the client and server machines. The *mount* command initiates a connection to a daemon (listener) process on the server machine using the transport interface. Once connected negotiation of run time

environment parameters is done. Included in this are the environment release version number, security parameters, and hardware architecture type. If the machines have heterogeneous machine architectures, an external data representation (XDR)^[4] is used to allow them to communicate. To avoid performance degradation XDR is only used when necessary. Once the negotiation is complete the virtual circuit is passed into kernel address space using a new *rfsys(FWFD)* system call. This virtual circuit establishment procedure moves the network connection complexity out of the kernel into user level routines and takes advantage of capabilities provided by existing software.

Client

A client process is a process that accesses a remote resource. Remoteness detection is dependent on the type of system call being executed. For pathname based system calls, remoteness is detected upon the traversal of a mount point of a remote resource (e.g., /foo in Figure 1). For file descriptor based system calls remoteness is detected when an RFS inode is encountered. The client implementation for these two cases is considered separately.

In the pathname case the system call uses the *namei()* and *iget()* functions to resolve the pathname to an inode. Specifically, *iget()* is used when traversing down into a mounted file system. The *iget()* function has been modified to pass control to the RFS module when traversing into a remote file system. Upon entering the RFS module the system call under execution is suspended. A request message representing the system call and the remainder of the pathname is sent to the server machine using the communication pointer from the RFS inode in the mount table. The client process then blocks until a response is received from the server.

For those system calls that require no further access of the inode (e.g., *chmod*, *chown*) control is returned directly (via *longjmp*) to the user, instead of through the system call routine which initiated the remote access. This allows RFS to do remote system calls without having to change each system call in the kernel, thus minimizing altered kernel code.

For those system calls which establish an inode which will subsequently be referenced (e.g., *exec*, *open*) control is returned to the initiating system call routine. Before control is returned an RFS inode is created using the communication pointer contained in the response message from the server. This RFS inode is then passed to the initiating system call routine.

The file descriptor case uses FSS for remoteness detection. A file descriptor for a remote file is the result of a pathname based system call (e.g., *open*, *create*) (Figure 8).

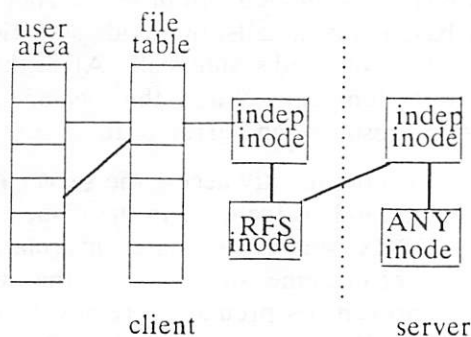


Figure 8. Open Remote File

The file descriptor is associated with an RFS inode through the local file table. In the course of executing the system call the FSS encounters the RFS inode. At this time control is passed to the RFS module. The RFS module uses the same request/response mechanism described above to complete the system call. Upon completion, control is returned to the initiating system call routine.

The *read()* and *write()* system calls require additional data transport. For the *read()* system call a read request message is sent to the server. The server satisfies the request by sending the requested data in the form of data messages to the client. For each data message received the client copies the data into the user supplied buffer. A lightweight protocol between the server and the client is used to handle flow control. In an effort to minimize network access the last data message is combined with the response message for the initial read request.

The write system call behaves similarly but data messages flow from client to server. For each block of data required to satisfy the write request, the server sends a data request message to the client. The client responds by sending the next block of data to be written. The first block of data is combined with the initial write request message, saving a data request message and a data message, minimizing network access.

Server

The server is a kernel process, scheduled like any other process. It has a user area, although there is no user instruction space, no bss, and no text space. The code it executes is solely in the kernel so there are no context switches between user and kernel mode.

The role of the server process is to receive request messages from client machines and execute them locally as if they were system calls that were initiated on the server machine. When the system call completes, the server returns the requested resource to the client along with any error indication.

The server is a transaction based process; each request is executed to completion before another is begun. Its sole purpose in life is to service requests from remote machines. Multiple server processes can and do exist on any machine that wishes to provide file service. Servers are not associated with any particular client machine or client process.

After receiving a request the task of the server is to masquerade as the requesting process. The request message contains enough information (e.g. uid, gid, ulimit) for the server to appear to the remote machine as the client process. Being a kernel process, the server can extract data from the request message and store it directly into its own user area.

Before fulfilling the request the server must recreate the client's environment on its own machine. This varies depending on whether this request was due to a pathname or file descriptor based system call. For a pathname based system call the pathname is set up from the request message. The pathname in this case is the pathname remaining beyond the remote mount point. The pathname evaluation will proceed from the inode of the advertised resource. For file descriptor based system calls, the inode specified by the RFS inode on the client side is used to complete the requested system call. After the environment is set up the server executes the specified system call. When the system call completes a response message containing the requested resource and error status is sent to the client.

In the cases where the client will subsequently access the server inode (e.g., *open()*, *chdir()*), a communication pointer to that inode is included in the response message to the client. In these cases, to preserve the UNIX semantics, state information reflecting the client's reference is recorded on the server machine. In particular, the inode reference count on the server inode is incremented to prevent its premature removal. Consider the case where a client process opens a remote file. If another process attempts to remove this file the inode will remain intact because the reference count remains high. If the reference count were not incremented when the client opened the file, the inode would be prematurely removed causing the client's file operations to fail, violating the UNIX file system semantics. In addition to inode reference counts, file/record locks and reader/writer counts for named pipes are also recorded on the server inode.

Since client state information is "recorded" on the server, a recovery mechanism for "erasing" this state in the event of a client crash was designed. The details of the recovery

mechanism are described below.

Recovery

The main purpose of the recovery mechanism is to restore state on the server machine in the event of a client machine crash, and to cleanup a client machine in the event of a server crash. Recovery is based on the existence of a virtual circuit between the two machine. The underlying transport provider signals the recovery mechanism when a virtual circuit breaks, indicating that the other machine crashed. RFS does not distinguish between a network failure and a crashed machine. The recovery procedure for clients and servers is different.

On the client side the recovery process wakes up any client process that is waiting for a response from the crashed server and marks RFS inodes indicating that the link went down, so that subsequent operations on these inodes will fail. It is important to note that a client process awakened by recovery returns an ENOLINK error to the user indicating the server machine crash. The client recovery mechanism also sends a message to a user level daemon which initiates a user level recovery procedure.

On the server side the recovery process "undoes" any state that the crashed client has recorded on the server. This is done by maintaining a per client record for each accessed inode (Figure 9). This record contains the number of references the client has to the inode. If the client machine crashes, the server knows how much to decrement the inode reference counts. These records not only contain inode reference count information, they also contain reader/writer counts for named pipes, so reader and writer synchronization can be restored in the event of a client machine crash. The server recovery mechanism also removes any file/record locks that a crashed client machine has placed on any of its files, to prevent other processes from blocking indefinitely. This is accomplished by recording the system identification of the client machine in the file/record lock structure when the lock is set. If a client machine crashes, all file/record lock structures with the system identification of the crashed machine are removed.

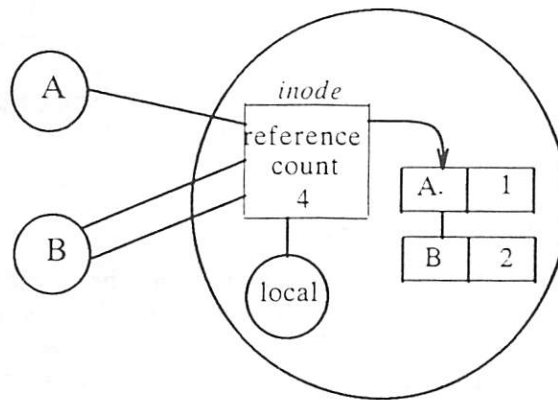


Figure 9. Server Recovery

5. Interesting Issues

In the course of the development several interesting issues were encountered. Some of these issues concerned special devices, time, and *stat()*. These issues are described below.

Special Devices

Special considerations had to be made for supporting special devices. Since the UNIX System treats special devices and named pipes as part of the file system most of the job was done. However, three problems had to be solved. First, slow devices and named pipes could consume all available server processes, making the server machine unusable. Second, when a client process sleeps waiting for I/O, a remote signal mechanism must be available to allow the client process to "break out" of the sleep. Third, remote data movement between the

server and the client must be transparent to the device driver.

The first problem was solved by using a dynamic pool of kernel server processes. If all free servers are busy and a client request is received, RFS creates a new server process to handle the request. The size of this server pool is limited by minimum and maximum tunable parameters. In the event that the pool reaches maximum size, the last kernel server is prevented from sleeping, which avoids a deadlock situation.

To solve the second problem the signal mechanism was extended to allow remote signaling. The first step was being able to uniquely identify a process within a distributed environment. This is accomplished by using a system identification (sysid) in association with a process identification (pid). The sysid uniquely identifies a particular machine, while the pid uniquely identifies the process within the machine. The remote signal mechanism is described below.

A client process that has sent an interruptible system call (e.g., *read()*,) request may sleep waiting for a server to complete the I/O operation. If the process receives a signal, a signal request message containing the client's pid and sysid is sent to the server machine. On the server machine, a server process services the signal request by using the sysid and pid to locate the server process sleeping on the I/O operation, and posts the specified signal to that server process (Figure 10).

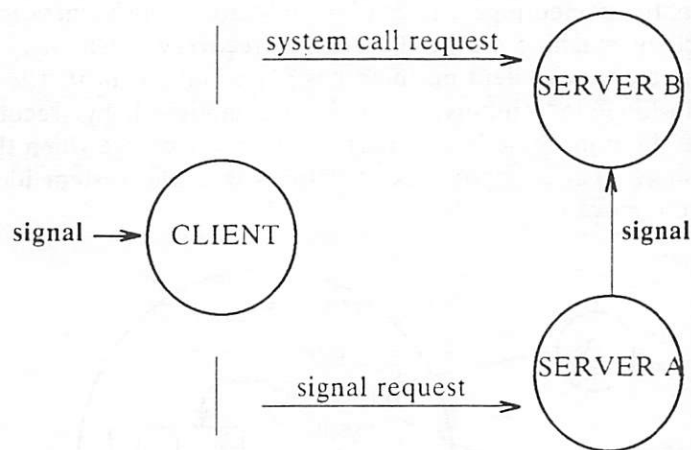


Figure 10. Remote Signal

To isolate the problems of remote data transfer from the device drivers, remoteness detection had to be done at a level below the device driver. The *copyin/copyout* routines are a standard interface used by device drivers to transfer data between kernel address space and user address space. By doing remoteness detection at this level, remote data movement is transparent to the device driver.

The *copyin* and *copyout* routines have been modified to check if a server process is attempting the data movement. This is done by checking a flag in the process' process table entry; a special flag has been reserved in the process table to differentiate between server processes and regular processes. If it is a server process, control is passed to the RFS module, which transfers the requested data to or from the client machine. Upon completion of the data transfer control is returned to the initiating routine.

In addition to data movement the *ioctl()* system call is greatly simplified by the *copyin/copyout* mechanism. A remote *ioctl* system call is passed to a device driver through a server process servicing an *ioctl* request. In response to the *ioctl*, the device driver may read or write data to a supplied address using the same *copyin/copyout* interface. As before the remote data transfer will be transparent to the device driver, making the remote *ioctl* implementation quite easy.

Time Skew

Time differences between client and server machines can cause an inconsistent view of file age. This may cause time sensitive commands such as *make*, *news* and *mail* to break.

This time skew problem was solved using a time delta approach. Upon establishing a connection between a client and a server, the time delta between the two machines is calculated and recorded. Time based information sent in response to a client request (e.g. *stat*) is adjusted using this delta to compensate for the inconsistency between the two machines. The time delta is recalculated when either machine changes its current notion of time.

stat()

In a distributed environment, where unique device numbers are not guaranteed, applications using *stat()* to obtain the device and inumber of files to check equality may break. To solve this problem the contents of the device field are modified for remote files. The most significant bit in the device field is used to indicate whether the file is local or remote. The next seven bits are used to record the server machine on which the file exists. The last eight bits are used to record the mount table index of the file system in which the file resides. With this scheme the device field will be unique over a network environment, making device/inumber comparisons work.

6. Conclusion

The design of RFS clearly exhibits AT&T's commitment to providing truly transparent file access without compromising the UNIX file system semantics. By maintaining client state information on the server machine to assure data integrity and consistency RFS can be used by existing applications with no fear of malfunction. Allowing access to remote special devices allows users to share expensive peripheral devices easily and economically. In all RFS provides a fully distributed file sharing environment.

7. Acknowledgements

Many people contributed to the ideas and spirit of the RFS project. The project was supervised by Art Sabsevit. The prototype system from which much of RFS was based, was done by Dave Arnovitz and Jeff Langer. We would like to thank Tom Houghton, Steve Buroff, Gil McGrath, Laurence Brown, Maury Bach, Her-daw Che, Mike Padovano, Anil Shivalingiah, Al MacPherson, and Ron Gomes for their help in designing and debugging the system. Also, we would like to thank Peter Weinberger of the UNIX Research Laboratory of AT&T Bell Laboratories for his help during the early stages of the project.

REFERENCES

1. D. M. Ritchie, "A Stream Input-Output System," *AT&T Bell Laboratories Technical Journal* 63(8) (October 1984).
2. D. E. Kevorkian, "System V Interface Definition" *Spring 1985 Issue 1*
3. D. J. Olander, et. al., "A Framework for Networking in System V" *USENIX Conference Proceedings*, Atlanta, Georgia (June 1986).
4. SUN Microsystems, "External Data Representation Protocol Specification" (April 15, 1985).

The Generic File System

R. Rodriguez, M. Koehler, R. Hyde
decvax!rr, decvax!koehler, decvax!rich

ULTRIX[†] Engineering and Advanced Development Group
Digital Equipment Corporation
Continental Blvd.
Merrimack, New Hampshire 03054

1. INTRODUCTION

The UNIX[‡] operating system is often compared to other operating systems. The UNIX file system often pales in contrast to other file systems. The UNIX file system has been called many things, including and, in no particular order: fragile, slow, and useless. Significant changes must be made to the UNIX file system to address the new world of networking, nonlocal file systems, and file servers.

Market pressure exists to support several types of file systems and disk formats. Specific architectures to support are the Fast File System (FFS) from the University of California at Berkeley, the Network File System (NFS) from Sun Microsystems, the VMS file system from Digital, the MS-DOS[§] file system from Microsoft, and the System V file system from AT&T.

To support a diverse set of file systems, we have created the Generic File System (GFS) Architecture. This paper discusses the work done within the GFS framework to fix some of the long standing problems in the UNIX file system and to support all of the file systems mentioned above, along with future file systems.

The specific goals of the present GFS project were to:

- Support multiple local disk formats through one kernel interface by decoupling the file system from the rest of the kernel
- Support NFS and future remote file systems
- Support both stateful and stateless file systems
- Keep the performance of GFS at least equal to the current file system
- Keep the reliability of GFS at least equal to the current file system

It was reasonable several years ago for Berkeley to evolve the old 1K block file system into the 8K block FFS. With advances in mass storage technology, a redesign of the FFS to support very large block transfers (64K or more) or special hardware (write once optical disks) would present major problems.

Without discarding the current file system, we want to support many hierarchical file systems through a single generic interface. This allows compatibility with existing file systems, and the ability to take advantage of new file system technology in both hardware and software.

The primary goal is to support all types of hierarchical file systems (local, remote, stateful, and stateless). This requires new functionality and rewriting of old code. The new structure of the file system allows the unfolding of loops and

[†] ULTRIX, MicroVAX, VAX, DEC, VMS and GFS are trademarks of DIGITAL.

[‡] UNIX is a trademark of AT&T.

[§] MS-DOS is a trademark of Microsoft Corporation.

streamlining of code that was not possible in the present file system.

File system reliability is difficult to achieve within the current UNIX kernel framework. Separating the file system code from the kernel, should make the file system more reliable and easier to maintain. We believe the file system reliability has been preserved or enhanced and its performance has increased.

2. GFS Design Philosophy

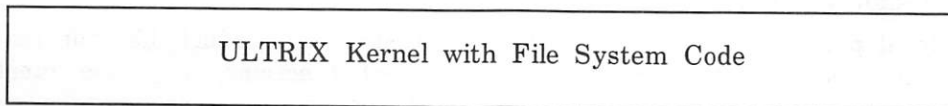
We designed the GFS architecture to support both stateful (like the Remote File System, RFS) and stateless (like NFS) remote file systems. We also had the goal of being able to evolve new file systems while maintaining the present ones. We did not create the GFS interface by copying or adding to existing interfaces, for example, the Sun Virtual File System Interface. The GFS interface, with its rich set of functions, will support stateful file servers and already supports stateless file servers (NFS).

Part of our design philosophy is that GFS controls all common file system resources. The mount table, the gnode table, and the buffer cache are all common resources that are maintained and allocated by GFS. Specific file systems must request these resources from GFS. GFS becomes the sole interface for file system operations and all specific file systems must communicate with GFS and not between each other.

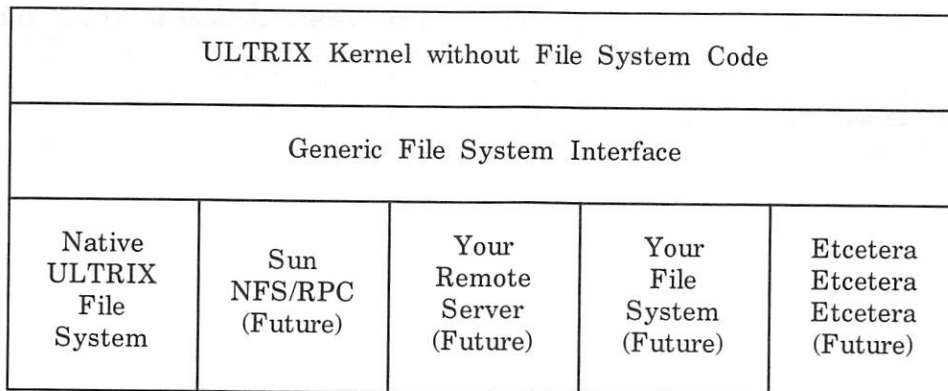
The major work in this project was to divide the existing file system implementation into a set of generic operations and a set of file system specific operations. This allows specific file system dependencies to be removed from the kernel. These generic operations were then expanded to communicate with multiple subordinate file systems, thus creating the generic file system interface to the kernel.

Below are two schematic diagrams which depict the current ULTRIX kernel and file system architecture (one amorphous blob) and how the GFS architecture insulates the kernel from any file system implementation.

Current ULTRIX File System Architecture



Generic File System Architecture



The old ULTRIX kernel contains file system code throughout many parts of the kernel. We had to make a very clean interface between high level file system operations and the kernel, decoupling kernel functions from file system operations. For specific file systems, we have created routines permitting an interface to all generic operations. This creates the generic file system interface to local file systems and network servers. The purpose of these generic routines is twofold. First, they simplify any interaction between GFS and the specific file systems and between GFS and the rest of kernel. Second, they eliminate any specific file system peer communication.

The design of GFS grew from four major factors:

- To support the local ULTRIX file system (UFS, really the FFS) and NFS. Stateful and stateless implementations must be supported. Local file systems have state no matter how it is hidden. Remote file systems may or may not have state depending upon the implementation. We fashioned GFS around these last two statements. We did not wish to exclude any possibilities with remote file systems and so we support both without prejudice.
- To increase reliability and maintainability. The file system insulation and reorganization was in itself a major enhancement for both reliability and maintainability. We redesigned the *mount* and *umount* interfaces to accommodate arbitrary local and remote file systems. The addition of the *getmnt* system call to clean up the access to mounted file system data was crucial.
- To maintain performance (and hopefully to increase it). By using indirect function calls, file system overhead would increase. However, the GFS architecture helped recoup these losses. This was probably the most difficult of our four factors to implement.
- To do binary distributions of new file systems, and for customers to create new file systems having a binary-only license. The design needed to support the idea that no kernel or user source code need change because a new file system was added.

3. GFS Design and Implementation

The focal point of this design is the mount structure. Much like the file table which contains pointers to appropriate file related functions, we have created a *mount_ops* structure that contains the functions that define a specific file system. The operations are bound to the mount structure at mount time.

All functions in the *mount_ops* structure are callable through macro interfaces. Operations that are not supported on a specific file system contain a *NULL* function pointer. When a non-existent operation is requested, the error *EOPNOTSUPP* is returned. A special function is needed to mount each different file system. This function is defined in a table at load time. At mount time, a file system type is passed in through the mount system call and the specific file system's mount routine is called.

The *mount_ops* structure includes the following functions:

```
struct mount_ops { /* begin mount ops */
/*      return value      function      arguments */
    int      (*go_umount)( /* mp, force */ );
    int      (*go_sbupdate)( /* mp, last */ );
    struct gnode * (*go_gget)( /* dev, mp, ino, flag */ );
    struct gnode * (*go_namei)( /* ndp */ );
    int      (*go_link)( /* gp, ndp */ );
    int      (*go_unlink)( /* gp, ndp */ );
    struct gnode * (*go_mkdir)( /* pgp, name, mode */ );
    int      (*go_rmdir)( /* gp, ndp */ );
    struct gnode * (*go_maknode)( /* mode, ndp */ );
    int      (*go_rename)( /* gp, from_ndp, to_ndp, flag */ );
    int      (*go_getdirents)( /* gp, uio, ctxt */ );
    int      (*go_rele)( /* gp */ );
    struct gnode * (*go_alloc)( /* gp, gpref, mode */ );
    int      (*go_syncgp)( /* gp */ );
    int      (*go_free)( /* gp, inode, mode */ );
    int      (*go_trunc)( /* gp, newsize, ctxt */ );
    int      (*go_getval)( /* gp */ );
    int      (*go_rwgp)( /* gp, uio, rw, flag, ctxt */ );
    struct filock * (*go_rlock)( /* gp, cmd, flino, filock, flock */ );
    int      (*go_seek)( /* gp, loc */ );
    int      (*go_stat)( /* gp, statbuf */ );
    int      (*go_lock)( /* gp */ );
    int      (*go_unlock)( /* gp */ );
    int      (*go_gupdat)( /* gp, atime, mtime, wait, ctxt */ );
    int      (*go_open)( /* gp, mode */ );
    int      (*go_close)( /* gp, flag */ );
    int      (*go_select)( /* gp, rw, ctxt */ );
    int      (*go_readlink)( /* gp, uio */ );
    int      (*go_symlink)( /* gp, source, dest */ );
    struct fs_data * (*go_getfsdata)( /* mp */ );
    int      (*go_fcntl)( /* gp, cmd, args, flag, ctxt */ );
    int      (*go_ioctl)( /* gp, cmd, data, mode, ctxt */ )
} *m_ops;
```

Having defined the mount structure and a generic set of operations on each file system type, different on-disk structures or network structures need to be accommodated. This gives rise to the *gnode* or generic inode. Because we no longer have a known on-disk inode format, we need to invent a way for the kernel to access the attributes it needs to use while allowing different formats of on-disk inodes. The *gnode* fulfills that need.

The *gnode* is an enhanced generic form of the old *incore* inode. Access to generic file attributes is required throughout the kernel. This is accomplished through the *gnode_common* structure (which is part of the *gnode*). It is the responsibility of each file system implementation to read its own disk inodes and fill in the structure appropriately. Likewise, when the kernel modifies the data in the *gnode_common* structure, the specific file system is required to use the structure to update its on-disk inodes.

Here are the *gnode_common* and the *gnode* structures:

```

struct gnode_common {
    u_short    gc_mode;      /* 0: mode and type of file */
    short      gc_nlink;     /* 2: number of links to file */
    short      gc_uid;       /* 4: owner's user id */
    short      gc_gid;       /* 6: owner's group id */
    quad       gc_size;      /* 8: number of bytes in file */
    struct timeval gc_atime;  /*16: time last accessed */
    struct timeval gc_mtime; /*24: time last modified */
    struct timeval gc_ctime; /*32: last time inode changed */
};

struct gnode {
    struct gnode_req {
        struct gnode *gr_chain[2]; /* must be first */
        ...
        struct mount *gr_mp;      /* where my mount structure is */
        ...
        union {
            daddr_t gf_lastr;      /* last read (read-ahead) */
            ...
            struct {
                int lastr;          /* lastr slot */
                struct mount *gm_mp; /* mounted on */
            } g_pmp;
        } gr_un;
    } g_req;
    union {
        char pad[128];            /* ULTRIX disk inode size */
        struct gnode_common gn;   /* the common data */
        struct {
            struct gnode_common _x;
            char *free;
        } _freespace;
    } g_in;
};

```

For the current ULTRIX file system, the on-disk inode is 128 bytes. The *gnode* contains the field *char pad[128]*, which is 128 bytes long. An ULTRIX disk inode can be read directly into this field because this field overlaps with the *struct gnode_common gn* field. Other file systems with a different layout would have to read their disk inodes in elsewhere and then set up the fields in the *gnode_common* area. If *gp* is a pointer to a *gnode*, then, *gp->g_in._freespace.free* is a pointer to 88 bytes (128 bytes - sizeof(struct *gnode_common*)) that is free to be used however the specific file system wants.

With the general data structures in hand, we need to take a step back and consider a few of the fine points. Remote file systems precipitate another new element in GFS. Remote file systems need the notion of *context*. Context contains data such as the current directory, the root directory, uids, and gids. Because of situations like *setuid* programs, occasions will exist when the context of the process that opened a file is not that of the process that is carrying out the operations on the file. Therefore, the context of the original process must be kept. Context is an attribute of every open file pointer, and therefore it is required for most of the GFS functions. All the specific file system routines must use this context structure and avoid using the user structure when servicing requests.

To support arbitrary disk formats and disk inodes, care has to be taken to simulate UNIX semantics in the file system, particularly for the directories, "." and "..". In general, a system call is needed to supply directory entries so that each local and remote file system can keep its directories in its own format, while returning entries in a generic directory structure. It is also required that each file system return the first two entries in a directory as "." and "..", and that they correctly simulate the UNIX semantics for "." and "..". The system call to obtain directory entries is called *getdirents*.

The file */etc/mtab* is inherently unreliable. A system call (*getmnt*) and generic data structure exist to eliminate */etc/mtab* and remove the file system dependencies from the kernel and the tools. */etc/mtab* is dead. The data structure *getmnt* returns is:

```
struct fs_data {
    u_short    fd_flags;           /* how mounted */
    u_char     fd_fstypes;        /* see ../h/fs_types.h */
    u_int      fd_gtot;           /* total number of gnodes */
    u_int      fd_gfree;         /* number of free gnodes */
    u_int      fd_btot;          /* total number of blocks */
    u_int      fd_bfree;         /* number of free blocks */
    u_short    fd_mtsize;        /* max transfer size */
    u_short    fd_otsize;        /* optimal transfer size */
    char       fd_devname[MAXPATHLEN + 1]; /* name of dev */
    char       fd_path[MAXPATHLEN + 1];  /* name of mount point */
    dev_t      fd_dev;           /* major/minor of fs */
    u_int      fd_bfreen;        /* user consumable blocks */
    u_int      fd_spare[100];     /* lots of room left, use it */
};
```

With the design worked out this far, we now had a large number of routines in the kernel and user tools to fix. One test of completeness for removing all file system dependencies was to remove all mention of the include file *fs.h* from the kernel source directory, */sys/sys*. Because of the new system call for directory entries, we also completely rewrote the library routines for directory manipulation, *directory(3)*. Only one program comes to mind that needed fixing because of this change: *restore*. *Restore* was recoded because it had its own internal directory routines.

Because accurate data about mounted file systems was now available through the *getmnt* system call, we fixed *mount*, *umount*, *fsck*, *df*, *mkfs* and a few other problem areas. The *mount* command was generalized to handle other parameters which were the file system type and an arbitrary option field for network mounts.

The format of the file */etc/fstab* had to be generalized to handle different types of file systems and their associated options. Enhancements to the kernel unmount code and shutdown code allow us to unmount file systems cleanly and set a byte in the superblock. *Fsck* recognizes this byte and will NOT check a clean file system needlessly. This enhancement probably got us more immediate praise than anything else because this is highly visible in our environment.

4. GFS Status

GFS is currently running in a production environment. It became a reality on March 26, 1986 when it went multi-user in a production environment. Over the initial week, we encountered some minor problems. The system has been very stable since these problems were corrected.

5. GFS Performance

Figures 1, 2, and 3 show performance data on an ULTRIX GFS kernel, an ULTRIX V1.2 kernel and a Beta 4.3BSD kernel. The tests for the RQDX2 and RQDX3 controllers were performed on a MicroVAX II with 3 Megabytes of memory. The disk was an RD53 and the timings were of a tight loop that sequentially wrote and read an 8 Megabyte file. For the UDA50 controller, the tests were run on a VAX 11/780 with 16 Megabytes of memory. The disk was an RA81 and the timings were of a tight loop that sequentially wrote and read a 16 Megabyte file. In each case, the procedure was repeated five times and averaged. The standard deviation was computed for each test and was always less than 1%.

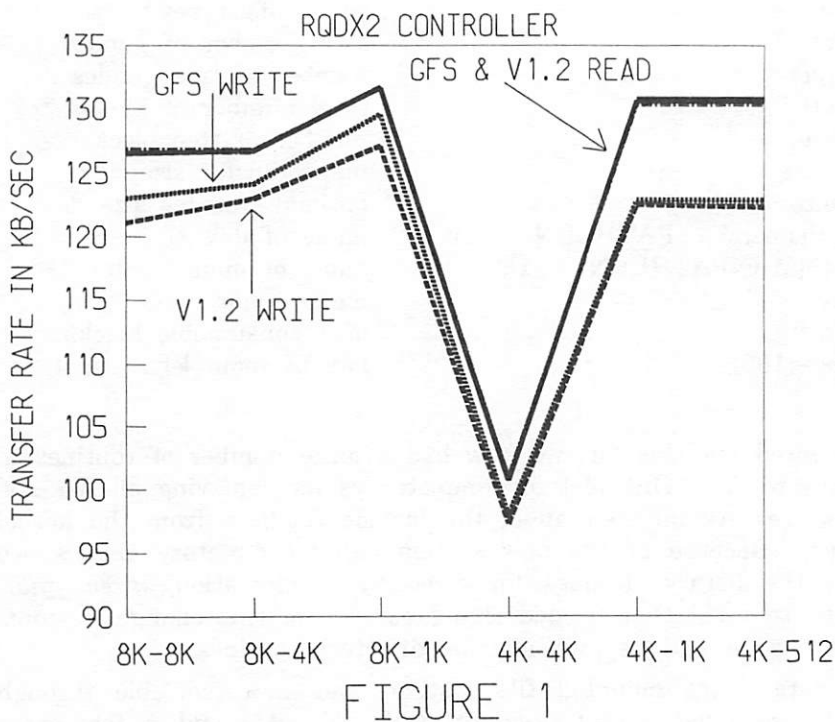


Figure 1 for the RQDX2 shows that both GFS and V1.2 give identical performance for reading a large file. GFS gains 2% on write speed for file systems with an 8K block size. The two systems are equal for a 4K block size.

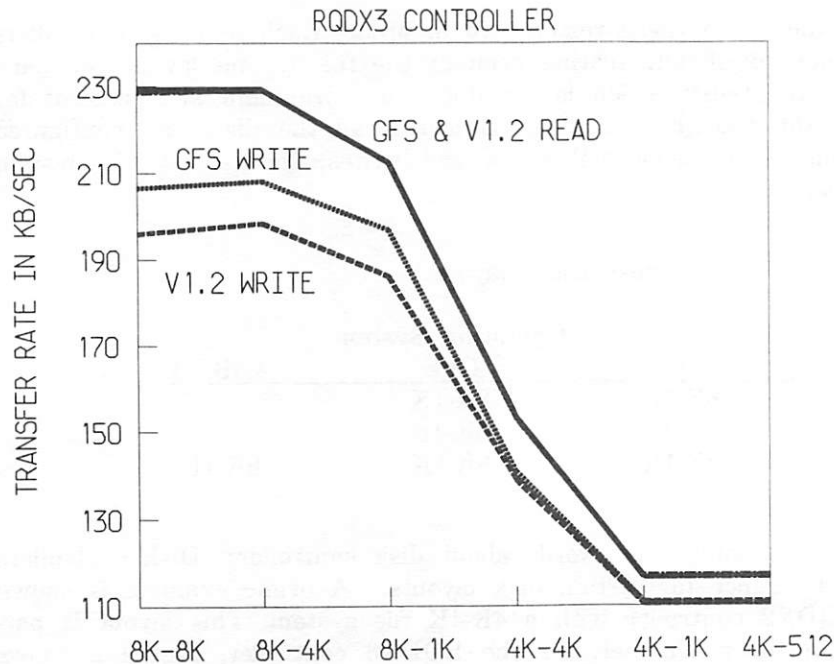


FIGURE 2

Figure 2 for the RQDX3 shows that both GFS and V1.2 give identical performance for reading a large file. GFS gains 5% on write speed for file systems with an 8K block size. The two systems are equal for a 4K block size.

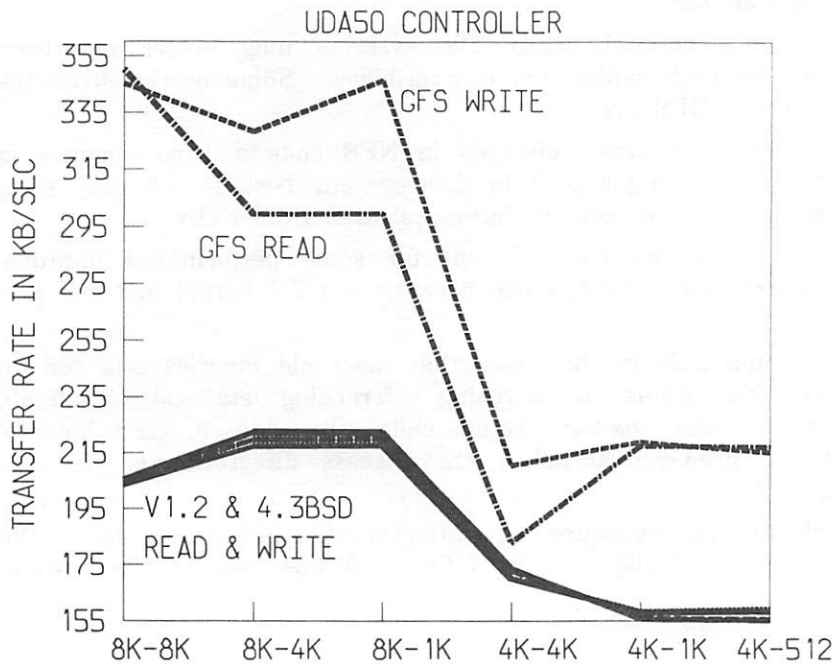


FIGURE 3

Figure 3 for the UDA50 shows that both 4.3BSD and ULTRIX V1.2 give identical performance for reading and writing a large file. GFS gains a large amount on the UDA50 controller. Most of this is due to a driver fix. We tested an ULTRIX V1.2 kernel with the driver fix and we found that the read speeds were the same but that GFS gained 2% on write speeds for all configurations except the 8K-1K file system where it gained 15%.

Some general comments on these results are in order. Each of these controllers is very different. Without specifically testing each of the file system layouts on each controller, it is difficult to predict which layout (block and fragment size) is best for each controller. In general, though, not much is lost if all the disks are configured with 8K-1K file systems. For sequential reads and writes, the fastest file system layout for each controller is:

Fastest Disk Layout

Disk Controller	Operating System		
	V1.2	GFS	4.3BSD
RQDX2	8K-1K	8K-1K	-
RQDX3	8K-4K	8K-4K	-
UDA50	8K-1K	8K-8K	8K-1K

Last but not least are some sage words about disk controllers. Disk controllers can have very poor performance for certain disk layouts. A prime example is shown in Figure 1 for the RQDX2 controller with a 4K-4K file system. This layout is particularly bad for this controller. Further, for the RQDX3 controller, Figure 2 shows that the bigger the transfer size, the better for this controller. It should also be noted that these tests do not show the total amount of work any of these controllers can do. Saturation tests need to be done to show how much total overlapped I/O the controllers can support.

6. GFS Strengths and Weaknesses

We believe a clean path through which file systems may evolve has been created. This path allows for both sanity and compatibility. Some of the strengths and weaknesses that we see in GFS are:

- GFS supports NFS with minor changes in NFS code and no changes in RPC code. This was a major goal in development because we may want our NFS implementation to support future releases from SUN.
- GFS is transparent to the user. Except for some performance improvements, users cannot tell the difference between a GFS kernel and the pre-GFS kernel.
- GFS is forward-compatible in the sense that most old binaries still run on the new system. Exceptions are anything referencing `/etc/mtab`, `/etc/fstab`, or using the `mount` and `umount` system calls. In addition, GFS binaries won't run on the pre-GFS kernel if they access directories or use the `getmnt` system call.
- GFS has had limited exposure to different file system types. We currently have one local file system, UFS, and one remote file system, NFS.

7. GFS Future

Plans for the future include adding new file system types. Some local file systems currently under consideration are the VMS file system from DEC, the MS-DOS file system from Microsoft, and the SYSTEM V file system from AT&T. Some remote file systems under consideration are the RFS from AT&T, the RFS from Brunhoff, and other efforts as they become available.

There are plans to prototype a non-UFS root file system in the near future. This will affect only the boot procedure.

There are plans to solidify the interface (as soon as a second local and a second remote file system are completed). We hope to publish that interface for use internally and externally.

8. Acknowledgements

We gratefully acknowledge the support of our management: Ken Olsen, Bill Heffner, Glenn Johnson, John Ferguson, and Jim McGinness. Discussions with Mike Karels and Kirk McKusick were very timely. Carolyn Zappala deserves our thanks for the tireless work she did in getting the performance numbers. Ricky Palmer and Larry Palmer get kudos for help in getting GFS and NFS to coexist and for reading the paper. The ULTRIX Documentation Group did another great job in helping to make this paper more presentable.

Above all else we owe a big (and we mean BIG) thank you to our understanding wives and close friends who tolerated us during this project. We also thank the cast of thousands that we cannot list in this short space.

9. References

The references that follow serve as good entry points into the literature. The interested reader would find many hours of enjoyable reading by tracking down these papers and those that they in turn reference.

- Brunhoff, T., "Design Considerations for Remote File Systems (Extended Abstract)", (unpublished but available on the USENET), 1985.
- Cole, Clement T., Flinn, Perry B., Atlas, Alan B., "An Implementation of an Extended File System for UNIX", USENIX Conference Proceedings, Summer 1985, p. 131-150.
- Kernighan, Brian W., Mashey, John R., "The UNIX Programming Environment", *Software Programming and Experience*, 9(1)p.1-15, January, 1979.
- McKusick, Marshall Kirk, Joy, William N., Leffler, Samuel J., Fabry, Robert S., "A Fast File System for UNIX", CSRG Technical Report 83-147, 1983.
- Popek, Gerald, et. al., "The LOCUS Distributed Operating System", *Operating System Review*, 17(5)p.49-70, ACM, October, 1983.
- Sandberg, Russel, Goldberg, David, Kleiman, Steve, Walsh, Dan, Lyon, Bob, "Design and Implementation of the Sun Network Filesystem", USENIX Conference Proceedings, Summer 1985, p. 119-130.
- Tichy, Walter F., Ruan, Zuwang, "Towards a Distributed File System", USENIX Presentation, Summer 1984.
- Weinberger, Peter J., "The Version 8 Network File System", USENIX Presentation, Summer 1984.

Modelling Text As A Hierarchical Object

James Waldo, Ph.D.
Apollo Computer Inc.
330 Billerica Rd.
Chelmsford, Ma. 01824
..decvax!wanginst!apollo!waldo

Abstract

In computerized typesetting and editing, text is most commonly represented using what we call the *linear stream* model. On this model text is represented as a linearly ordered series of codes that are interpreted by reference to a character set. The linear stream model has the advantage of being easy to understand and also lends itself to certain efficiencies of space. However, we will argue that the linear stream model is inadequate for the representation of text that contains both the characters that make up the text and information about those characters. As an alternative we will offer a model that treats text as a hierarchically organized set of objects in which only the base objects refer to a linearly ordered series of codes. This hierarchical organization is used to represent structural information about the text. Additional information is captured by associating properties with the objects. By using this representation of text in which a clear distinction is made between the characters making up the text and the information about the text, a number of problems inherent in the linear stream model disappear, and new ways of organizing and interacting with text become possible.

The Linear Stream Model of Text

Computer applications generally represent text using what we call the *linear stream* model. Let us begin by making precise what we mean by the linear stream model of text. A *linear stream text object* is defined as a five-tuple $\langle C, S, R, L, f \rangle$ where

- 1) C is a set of codes c_1, \dots, c_n ;
- 2) S is a sequence s_1, \dots, s_m such that for all x in S , $x \in C$;
- 3) R is a simple linear ordering relationship on the members of S ;
- 4) L is a set of characters k_1, \dots, k_m ;
- 5) f is a function from members of C to members of L .

It is easy to find examples of linear stream text objects. Most any text file using, say, 7-bit ASCII appears to be a suitable candidate. For such a file, C is the set $\{0, \dots, 127\}$, S is the sequence of codes in the file, R is the relationship of order in the file, L is the ASCII character set, and f is the function which maps the codes to the characters.

There is much to be said in favor of the linear stream model of text. One of its strongest features is its simplicity. Since the only relationship defined on the stream of codes is that of a simple linear ordering, linear stream text objects are mathematically straightforward and their properties are reasonably well understood. The independence of the character set and the code set also allows a reasonable amount of expressive power for the model. Using the model, we can adapt to various national character sets without a change in the model, but simply by changing the set L of characters and the mapping function from codes to characters f . The linear stream model also allows us to tailor the size of the character set and code set to meet our needs. If our text only needs to draw on a character set of 100 characters, we can represent the characters

using a seven bit code set, but if our character set grows to where it needs a larger number of distinctions, we can change the size of the code set without changing the underlying model of what a text object is.

But is this model really rich enough to allow us to represent text in an adequate fashion? A quick look at the ASCII character set table should make one suspicious, for there are some rather odd "characters" there. Things don't look bad if we confine our attention to that part of the table containing the "printable" characters, i.e., those characters with codes between hex 20 and hex 7e. Outside of that range, however, there are some very odd looking characters, perhaps the most understandable of which are "tab", "carriage return", or "line feed." One begins to get a feeling of ontological angst when one starts to think of these as characters, for surely "carriage return" is not the same sort of beast as the letter "a". Such angst may well turn to dread if we begin to consider some of the other characters on the low end of the table such as "form feed" or "ack". These hardly seem to be the same sorts of entities as the letters of the alphabet.

Our glance at the low end of the ASCII table serves only to remind us of what we really knew all along. The linear stream model of text, taken in its pure form, is simply inadequate as a representation of real text objects. Text is not just a linear stream of character codes that are interpreted as characters. Text is broken up into lines, immediately giving text a two-dimensional nature. Lines of text are organized into paragraphs that start with the first line indented, or as blocks of code, where each line is indented. In more sophisticated text objects, the characters may have several different representations (fonts), or there may be a mixture of character sets.

It is a testament to the seductive power and simplicity of the linear stream model of text that, in the face of such problems, the model was not discarded long ago. Instead, the model has been extended in two fundamentally different ways in an attempt to overcome its shortcomings.

The first extension is the one which we observed in looking at the low end of the ASCII table. By opening up the notion of "character" to include such things as "tab" and "form feed" the linear stream model can be extended to deal with much of what is needed to produce simple text from linear streams of codes. Just because such code interpretations are called characters, however, does not make them characters. Indeed, it is important to realize that such codes are interpreted in a fundamentally different way than, say, hex 61. Whatever is reading the text stream will interpret hex 61 as the letter "a", but the interpretation of hex 0c is a directive that tells whatever is doing the reading of the stream where to place the next character read. What we find in the low end of the ASCII character set is a collection of codes which are interpreted as the most common formatting directives. Thus, these codes allow us to retain the illusion of the simple linear stream model and yet still deal with simple cases of real text.

Such additions to the character set do not allow more sophisticated text manipulations such as changing the font used to render the characters, or changing the character set itself. To extend the linear stream model of text to encompass these notions requires the introduction into the character set of a special character, which because of its most common instantiation we will call the "escape" character. The purpose of the escape character is to act as a flag, telling whatever is reading the linear text object that what follows is not to be interpreted as regular character codes, but rather as special directives that are outside the scope of the character set currently in use. The "escape" character, of course, need not be the escape character (i.e., ASCII hex 1b) or, for that matter, even a single character. The flag could be the code for a particular character (say, the code for ".") immediately preceded by some other character

(say, the "carriage return"), and the amount of the stream to be interpreted as directive rather than as part of the object text is open to implementation decisions, ranging from a single code to the sequence of codes up to some specified character that acts as an "escape" escape.

The use of such mechanisms to extend the expressive power of the linear stream model results in what we call "mutant text streams." These text streams are mutant in that the notion of character code and character set have been extended to include not only those items in a text object that are actually rendered on the printed page or written to a screen, but also items that indicate how those items are to be rendered and how the subsequent codes in the stream are to be interpreted. More succinctly, mutant text is the result of using the same set of codes to represent both the characters which make up the text and information about the text. Mutant text is the dominant form of textual life in the world of the computer, encompassing everything from straight ASCII files to the source files used by sophisticated text formatters like `nroff` and `troff`.

Unfortunately, the use of mutant text objects involves us in problems of both a theoretical and a practical nature. On the theoretical level, the use of mutant text rests on a mapping of separate ontological entities into the same representation. Characters, information concerning how those characters are to be formatted, and structural information concerning relations among those characters are all mapped to a code or sequence of codes of the same type.

If this theoretical problem were the only drawback to the linear stream model, of course, there would be little cause for alarm; ontological considerations are generally secondary to the fact that something works. However, this conflation of different sorts of entities leads to a set of practical problems that are not so easily dismissed.

The most obvious practical problem with objects containing mutant text concerns the interpretation of a random character code within such an object. In a pure linear stream model text object, the interpretation of any character code is determined as a context free function of the code, the character set, and the mapping function from the code set to the character set. With mutant text, however, the interpretation of any code in the text object is inherently context sensitive. When interpreting a code in a mutant text stream, one must first determine if the code to be interpreted is part of the stream that is functioning as a directive or part of the stream that is to be interpreted as a character. Even if we know that the code in question is not part of a directive, since it is possible to change the character set and mapping function from codes to characters via a directive, one can not know from a randomly chosen point within a mutant text object what the interpretation of a code is without first scanning all of the codes prior to the code of interest.

One way of avoiding such a problem, of course, is to require that mutant text objects always be read sequentially, from beginning to end. If one can be assured that this is the only access method used on the text object, one can be sure that the information needed to interpret a given character code will always be available when that code needs to be interpreted. But such an approach, even if we could adhere to it, seems overly restrictive. This way out requires that we process text in an essentially batch-oriented mode, rather than allowing interactive access to full fledged text objects.

These extensions and restrictions on the linear stream model only serve to mask the underlying problem with the model. Simply stated, the problem with the linear stream model is that it represents both the characters making up a text object and information about the text object itself in the same way. Sticking to this model puts us in the position of having to choose one of two unacceptable alternatives. If we do not restrict the amount of information about the charac-

ters that can be contained in a text object, parsing of the linear stream becomes so complex that we are unable to process such objects at interactive speeds, thus consigning text applications to the era of batch processing. To gain speed, we can simplify the parsing task by restricting the types of information about the text that may be contained in the stream. In doing so, however, we limit the text applications to a subset of the possible documents with which we may wish to work. Neither of these choices is acceptable; what is needed is a new representation of text.

The Hierarchical Object Model of Text

We became aware of the limitations of the linear stream model of text during the initial design phase of the Text Management Library, a toolkit for constructing text applications currently under development by the User Environment group at Apollo Computer, Inc. The Text Management Library is intended to supply all of the functionality needed for any text application, from simple plain-text presentation to fully-interactive, intelligent document editors and structured program editors. Since applications built with the toolkit may deal with any sort of text, we were unable to restrict the information about the text contained in the text object in any way that could be captured by simply reinterpreting a small subset of the character set. Since the toolkit was designed to aid in the development of interactive applications, we could not assume that access to the text object would always be sequential. Hence we concluded early in the design that the linear stream model of text, even with extensions, was inadequate for our purposes. Instead, we decided to model text objects using what we call the *hierarchical object* model of text.

The first step in arriving at this model for text objects was to take the notion of *structure* in text seriously. Text is rarely a simple stream of characters. A document such as this paper is, at bottom, a stream of characters; but those characters are grouped together to form sentences, which are in turn grouped together to form paragraphs, which in turn are grouped together to form the final object which is the paper. More complex documents might have structural elements such as sections (made up of paragraphs), chapters (made up of sections), and volumes (made up of chapters). Source files are also made up of a base of characters, but these characters are organized in a different fashion — into lines that make up blocks that make up functions, etc.

The pattern is fairly obvious; beyond being a simple stream of character codes, text is really a hierarchically structured series of objects. At the lowest level, those objects refer to linear streams of codes. At higher levels, the connections between the object and the characters that make up the object are indirect at best.

When speaking of the structure of a text object, it is important to distinguish between the structure inherent in the text and the structure imposed on that text due to a particular way of formatting it. It is easy to confuse the two, as we rarely see text in an unformatted form; it is also difficult to draw a clean line between the two sorts of structure. The distinction is perhaps best seen by means of an example. In a document such as this paper, a basic element of structure is the paragraph. Any time I look at the paper, I can also see a basic element that makes up the paragraph which is the line. This latter element, however, is an artifact of the way I have formatted the paper. If I change font size, the characters making up an individual line may change, even though I have not changed the paper itself. However, changing the font will not change what constitutes a paragraph in the paper. The structure we are interested in is of the former sort which, we claim, is inherent in the text; not the latter, which is an artifact of the way the text is presented.

To reflect this kind of inherent structure, we decided to represent text objects using a generalized threaded tree structure in which the terminal nodes of the tree would refer to linearly ordered streams of character codes. Non-terminal nodes serve the purpose of imposing structural order on the text, but do not refer directly to character codes. Instead, the nodes of the tree are treated as objects that carry information about the text.

This notion of structure is not in itself new; indeed, it is similar both to the notion of a hypertext organization [4] and the organizational principles used in various "structure-based" editors [3]. Indeed, if one follows the simple definition of "hypertext" as "nonsequential writing"¹ hierarchical text objects are examples of hypertext. The structure of a hierarchical text object, however, plays a far more central role than is usual in other text systems based on the hypertext notion, in that the basic ordering of the characters in a hierarchical text object is determined by the structure, which may also be used for purposes of referring to other sections of the text object or to other text objects.

The treatment of text as a hierarchical object differs from that in structure-based editors in two ways. The first is that the hierarchical object model is more general. No interpretation is inherent in the various structures; all that is assumed is that the text is to be structured in some way. The second difference is that the structure in the hierarchical object model is taken to be an essential part of the text object. The structure tree is present both in memory and on disk. The text is never stored in a linear form. The unity of representation in memory and on disk is made easy on the Apollo system because of the ability to map disk object directly into a process' address space.

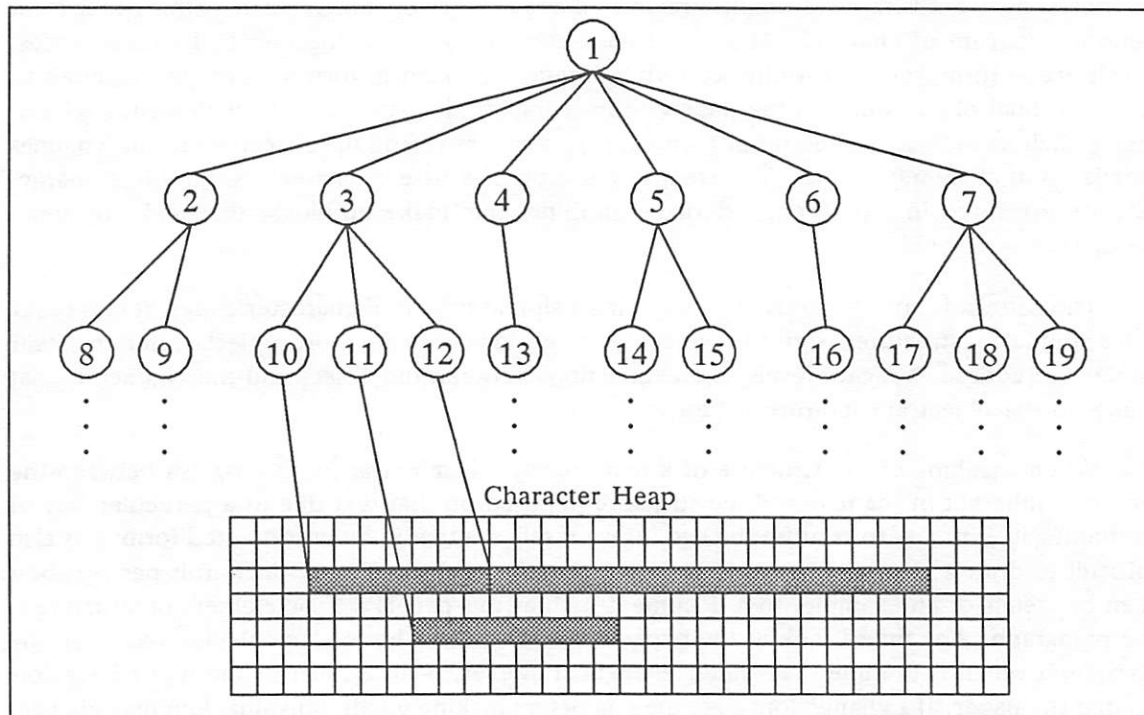


Figure 1. Partial Diagram of a Generic Hierarchically Structured Text Object

Part of the information carried by such a text object is inherent in the organization of the tree. For example, it is the configuration of the objects within the tree that determines the linear ordering of the characters that are pointed to by the terminal constituents of the tree. Character codes themselves never appear in the tree. Instead, the characters that make up the object are

stored in a single heap, and terminal nodes in the text structure point to areas in the heap that contain the characters dominated by that node. Note that the global ordering of characters within this heap need not have any relationship to the ordering of the characters in the text object. Linearity is maintained within the sub-areas of the heap pointed to by a particular terminal node, but no ordering is maintained outside of such sub-areas. This character heap can be thought of as a sort of primordial character goo, in which order is maintained only in very local areas. It is the hierarchical object structure of the text tree that provides global form and structure to the text object.

It should also be noted that the character goo is free of any character codes that could be considered mutant text. All codes within the character goo are interpreted as characters that will appear in the final rendition of the document, and none of them serve as directives concerning the placement of the characters that follow. A partial diagram of an example of a generic structure of this sort (generic because we have supplied no interpretation for the objects in the structure) is shown in Figure 1.

Treating text as a structured object also allows us to group text in ways not possible using the linear stream model. Since the linear stream model has only the ordering relation on the text, the only possible way of identifying a part of the text is by specifying a range of characters. Treating text as a hierarchically structured object, however, allows identification either by range or by structural entity. This allows the application working on the hierarchically structured object a range of identification granularity, from fine-grained identification at the level of individual codes in the heap to successively coarser grained identifications of objects higher up in the structural tree. In the structure shown in Figure 1, for example, we could specify a range of characters by giving the node and offset into the heap from the beginning of that node of the first character in the range (say, the fourth character dominated by node 10) and the node and offset of the last character. If the last character in the range were, say, the tenth character in node 12, the range would include all of the characters dominated by node 11. On a higher granularity scale, we can identify all of the characters in the heap pointed to by node 10 simply by referring to that node; all the characters pointed to by nodes 10, 11, and 12 by referring to node 3; and all of the characters in the object by referring to node 1.

A welcome outgrowth of treating text as a hierarchically structured set of objects is that it both expands and simplifies the set of possible modifications that can be performed on the text object. In a text object built using the linear stream model of text, the only interactions possible are those of inserting characters into the stream or deleting characters from the stream; more complex interactions are possible, but they ultimately reduce to some combination of the insertion and deletion of characters. Further, since the linear stream model of text requires that a global ordering relationship be maintained on the characters that make up the object, insertion and deletion require the reordering of all characters in the stream that occur after the point of insertion or deletion.

Using the hierarchical object model of text, no such global reordering is necessary during insertion or deletion of characters. Since the global ordering of the characters that are the lowest level constituents of the text object is determined by the structure of the object tree, inserting characters is accomplished by adding the characters to be inserted at the end of the character heap and then inserting a terminal node that points to those characters in the appropriate part of the structure tree. In the worst case this involves splitting a previously existing node into two nodes and inserting the new node between them. Deleting characters simply involves altering the starting position and/or the character counts of terminal nodes in the tree. Moving characters from one position to another in the text object does not involve any change in the character

heap; instead, the move is accomplished solely through moving the objects in the structure tree that point to the characters to be moved.

The structure shown in Figure 1 could naturally arise from an insertion or move. We can imagine a previous state of the structure tree in which node 10 pointed to all of the characters now pointed to by nodes 10 and 12, and in which either node 11 occupied some other position in the tree or had not yet been created (and hence the characters pointed to by that node were not yet entered into the heap). Suppose now that we wished to insert some characters between two of the characters pointed to by node 10. We would split node 10 into two nodes, adjusting the count of characters pointed to by node 10 to one edge of the point of insertion and creating a new node, node 12, pointing to the beginning of the characters forming the other edge. We then create a new node, node 11, positioned between nodes 10 and 12, and add the inserted characters to the end of the heap. Node 11 will point to those characters. The operation is similar if node 11 is being moved—we split node 10 and position node 11 between the resulting nodes. In the case of copy, of course, we do not need to operate on the heap, as the characters pointed to by the nodes are already entered into the heap. This method of accessing and manipulating the characters in a text object indirectly through the object structure is similar to the strategy used by the Lilith document editor [1] and the Bravo editor [2].

The set of possible interactions is naturally expanded by the hierarchical object model of text in that we can now base interactions on the objects in the structure tree as well as the set of characters. Objects that make up the structure tree may now be inserted, deleted, moved, and copied. At first glance, this may seem to be no more than a handy way to refer to blocks of characters that make up the terminal nodes of a sub-tree that has as its root the object being operated on. To see that there is more to it than this, however, one need only notice that we need to make a distinction in the hierarchical object model between copying an object and replicating that object. If we replicate an object in a new location in the structure tree, we add no new text to the character heap; we simply add an object pointing to the sub-tree being replicated to the appropriate spot in the structure tree. Such an operation gives us the same object appearing in multiple positions in the text object, as any changes made to that object will appear in all of the places in the document where that object occurs. Copying an object, on the other hand, creates a new sub-tree with the same structure as the original at the indicated point in the structure tree, and adds new characters to the end of the character heap that are pointed to by the terminal nodes of the new sub-tree. Such a copied object exists separately from the object from which it was created, and may be altered without those alterations appearing in the original object.

Properties of Objects

Since the toolkit must be sufficiently general for applications working with any sort of text, the Text Management Library does not presuppose any interpretation of the structure-inducing nodes other than that which arises out of their position in the threaded tree. The objects in the text tree by themselves impose order in that some are parents or siblings to others, but the effect of this imposed order is not inherent in the objects themselves. The objects that make up the nodes of the structure tree and that impose order on the characters in this model are themselves featureless containers relating only among themselves and whose influence can only be seen indirectly through the properties attached to them.

Determining how the text is to be presented is one of the functions of the properties that are associated with the objects in the text structure tree. It is the properties that give real meaning to the objects that make up the structure tree, and that take over much of the role that the

linear stream model of text relegated to mutant characters. Properties are not confined to determining how the text is to be formatted. Properties can also be applied to objects to convey information about the role of the object within the model itself, or to convey information about how the text dominated by that object is to be rendered.

The relationship between properties and objects is fairly straightforward. Associated with each object in the text structure tree is a set (perhaps null) of pointers to properties. At any time, a given property may be either active or inactive; the state of a particular property is determined by the current position in the structure tree. During traversal of the tree, properties are made active upon entry into a node that is made up of an object that contains that property as part of its property set; the property becomes inactive when the traversal mechanism exits the node containing that object.

Because of this mechanism, properties are inherited. Once a property is active it remains in force for all of the objects in the sub-tree for which the object having that property is the root. The only exception to this is in cases where the property is explicitly over-ridden by a property of an object which occurs as part of that sub-tree. If a property is over-ridden, it remains inactive only for the sub-tree whose root is the node having the property that did the over-ride. On exit of that node the property that was over-ridden is once again made active.

To see this more clearly, consider again the structure diagram in Figure 1. Suppose that node 1 has the property "having the font helvetica 18", that node 3 has the property "having the font helvetica 12", that node 11 has the property "having the font helvetica 12 italic", and that these are the only properties dealing with fonts in the text object. Upon entry into the tree, the property "having the font helvetica 18" is made active, and remains active until node 3 is entered. At this point, the property "having the font helvetica 18" becomes inactive, and the property "having the font helvetica 12" is made active. This property remains in force until entry of node 11, when the property is made inactive and the property "having font helvetica 12 italic" is made active. On exit of node 11, that property is made inactive and the property "having font helvetica 12" is once again made active. This property remains in force until exit of node 3, at which point that property is inactive and the property "having font helvetica 18" is again made active; this property remains in force everywhere else in the structure.

Our implementation of properties treats them as a pair of functions with associated argument vectors. The two functions, either but not both of which may be null, act as functions from the argument vectors to states of the text management library. One of these functions is the activation function. When combined with the first of the argument vectors provided in the property this function changes the state of the text management library into one in which the property is active. The second function has the responsibility of returning the state of the text management library to what it was prior to the execution of the activation function. Calling this function with the appropriate arguments has the effect of making the property inactive.

The toolkit supports two separate classes of properties. The first, which are called system properties, are supplied as part of the toolkit. Knowing full well, however, that we will not be able to predict all possible uses of the toolkit, we also provide support for a separate set of user-defined properties. Such properties have the same form as the system properties; however, the function table in which the entry and exit functions for such properties reside is available to the application. Thus the application developer is able to write his or her own entry and exit functions that may be loaded into this table, create properties that access these functions, and thus add those properties to the repertoire of the toolkit.

Perhaps the most obvious use for properties is to specify formatting information for the text object. Properties are used to determine the size of the area in which the text is to be formatted, the font in which the text is to be presented, and the setting of tab tables. Properties are also used to replace the non-character characters of mutant text streams. For example, there are no character codes that are interpreted as "tab" or "carriage return". Instead, a property is associated with an object that causes the text dominated by the object to be formatted with a given amount of leading space (for the tab property), or causes the next character to be placed on a new line (for the end-of-line property).

Properties are also used to store information about the way the text is to be rendered. Setting the foreground and background color to be used in painting the characters, for example, can be done by associating a property with a structure object.

Finally, properties can be used to convey information about the text model itself. Objects can be named via a property, and relationships between parts of the text object may be reflected with properties. Thus, even though there is no object that is inherently a paragraph in the toolkit, an application being developed as a structured document editor using the toolkit could create a property "being named a paragraph" and associate that property with certain objects. Having established such a property, the application could further require that certain other properties are always associated with objects having that property (such as being formatted in a certain way), or that only objects with certain properties (such as, for example, the property of being named a sentence) could be children of such an object.

Another set of properties that convey information about the text model are those that identify the character set to be used in interpreting the character goo dominated by an object. Using this method of changing character set allows this model to escape from the need to scan all characters prior to a given character to establish the interpretation of that character in cases of text objects in which more than one character set is used. The interpretation of any given character is still context sensitive, but the context has been reduced from all of the characters that precede a given character to the set of structure tree objects that dominate the character in question. Since the hierarchical text object is organized as a threaded tree, it is only necessary to traverse up the tree to the first object that specifies a character set to determine the interpretation of a given code.

To see how the property mechanism works, let us consider a few examples. One of the uses of properties is to determine or alter the font being used to present the text. This property has an effect on both the formatting of a text object (as the number of characters that will fit on a line will vary from font to font) as well as the rendering of that object (as the font used determines the shape of the characters that are output). As stated above, a property has the form of an index into a table of functions and a pair of argument vectors. In the case of a font property, the entry argument vector has a single member that is the name of the font, while the exit property vector is empty. Upon entry into an object in the structure tree with such a property, the font property entry function is called. This function pushes an identifier of the current font onto a local stack, and changes the current font for the text management library to the font identified by the entry argument vector. When this object is exited, the font property exit function is called, which pops the local stack to obtain the identifier of the previously current font, and re-instantiates that font as the current font.

As a second example, let us consider the property that replaces the mutant character "carriage return." Once again, the property consists of an index into a table of pairs of functions and a pair of argument vectors, but in the case of this property both entry and exit argument vectors

are null as is the entry in the function table for the entry function. Upon entry into an object in the structure tree with this property, the property is ignored. Upon exit of this object, however, the one function associated with the property is called; this function signals the toolkit that the line currently being formatted is full and that the next character should begin on a new line.

Since the property list associated with any object in the tree is made up of pointers to properties, it is possible for objects to share properties. For many properties, this will be the normal mode of association — there is no need for more than one “carriage return” property that all objects having that property point to. An application using the text management library may wish to have multiple copies of other properties, however, as it is possible to manipulate properties in the system. A font property, for example, may be interactively changed, as applications have access to routines that can be used to alter the argument vectors of a property. Changing a font identifier in a font property will change the font used to format and render all of the text dominated by objects that share that property. Such a global change may be just what the application wants. However, local changes in font may also be made by making multiple copies of a font property that may be altered without effecting other objects that have (copies of) that property.

Properties may be either absolute or relative. An absolute property is one interpreted without reference to the state of the properties at the time the property is active. An example of such a property would be “having the font helvetica 18 bold”. A relative property, on the other hand, can only be interpreted in the context of the properties already active at the time the property is encountered. An example of a relative property would be “having an italic face” which can only be interpreted relative to the font currently being used.

Since properties are full fledged entities within the text object, they may be accessed to gain information about the text itself. This fact may be used to exploit new methods of searching for text based not only on pattern matching within the characters that make up the text, but also on properties of the text object. Suppose, for example, that we wished to search backwards from a particular point in a text object for a certain pattern of codes that were mapped to a Cyrillic character set and output in blue. To perform such a search on a text object that was represented using the linear stream model of text would require that we search backwards in the code stream until we found the directives that caused the text to be rendered in blue and the character set to be Cyrillic; once those directives were found we would have to scan forward to find the directives that changed either the rendering color or the character set; and having found those limits the characters in between would have to be scanned in reverse order for the requested pattern. If the pattern were not found, we would have to scan the character stream to find the next point at which the directives for blue rendering and Cyrillic character set occurred and repeat the forward and backward search.

The hierarchical object model of text representation simplifies this process considerably. To perform such a search, we need only do a reverse traversal of the structure tree. Any terminal node that we come across where the properties “rendered in blue” and “mapped to Cyrillic character set” are not active may be ignored. In addition, the limits of the search on nodes that are encountered when those properties are active are well defined as all and only those characters dominated by the node. The advantages of such a search technique are that only characters that have the desired properties need be scanned for pattern matching, and all such characters are easily identifiable by the property structure of the text object.

Models, Conceptual Pages, and Rendering

The distinction made above between formatting properties, rendering properties, and modelling properties is hardly accidental; indeed, this three-fold distinction in the properties reflects a basic division of labor in the toolkit. The toolkit contains separate modules for manipulating the text model, for formatting that model, and for rendering a formatted version of that model.

This division of labor is made possible by the decision to reflect only the structure inherent in the text itself in the hierarchical text object, and not the structure placed on that text by the formatting. By making such a separation, the text object may be considered to be format-independent. Thus changing the format of such a text object does not change the basic underlying structure of the object itself. The distinction between the formatted version of a text object and the rendering of that object is a way of making the text object output-device independent. The rendering component takes as input a formatted text object, and translates that formatted object into the particular visual image appropriate for the current output device. If the text is being output to a printer, the rendering of the formatted object will look as much like the ideal formatted object as the resolution of the printer and the available fonts allows. However, if the output device is a screen, a number of options are available to the application—the screen (or the current window on the screen) may be treated as a viewport on to the formatted text, thus showing only a fraction of the ideal output, or the text object may be reformatted on the fly to fit into the output area available on the screen.

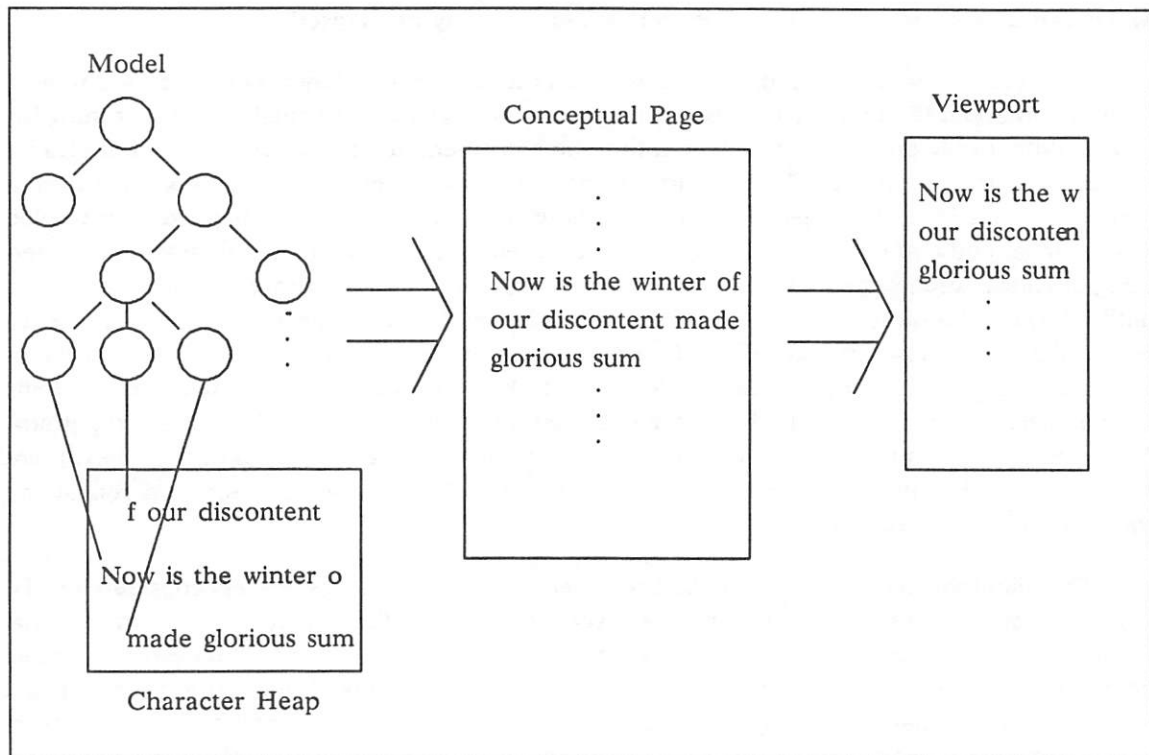


Figure 2 Mapping from Model to Conceptual Page (Formatter) to Viewport (Renderer)

The division is reflected not only in the operations performed by the various modules but by the basic entities on which those modules operate. The modelling component is fundamentally concerned with the hierarchical text object and operates on the tree structure of the ob-

jects, the properties associated with those objects, and the character heap. The formatting component translates this hierarchical text object into an abstract two-dimensional representation. The basic object in this representation is the conceptual page. A conceptual page has a width and a depth; it is on such conceptual pages that the text is laid out according to the formatting information contained in the properties present in the text object. It is the job of the rendering component to translate these conceptual pages into images, either printed or painted on a screen. To do this, the rendering component maps the conceptual page into a viewport, which may be larger, smaller, or the same size as the conceptual page.

Interactions may take place on any of the three levels of the toolkit. Changes to the model may alter the structure tree, add to that tree and, perhaps, the underlying character heap, or alter the properties associated with the objects in the model. Such changes will be permanent (unless undone) and will be immediately reflected in the view of the text object if they alter the appearance. Temporary changes to the formatting or rendering of the text object may also be accomplished by interactions with those components; such changes will not be saved in the object and hence are local to the particular editing or viewing session in which they are made.

Current Status of the Text Management Library

A preliminary version of the text management library, built using the hierarchical object model of text, is currently in use at Apollo. The library allows the construction of structure trees of arbitrary complexity, and includes a library of system defined properties that allow changing fonts and character sets, setting and resetting the size of the conceptual page, enabling and disabling of word wrap, and assorted other formatting properties.

Two editor development projects are using the text management library. The first of these is a plain-text editor, supporting the usual input, cut and paste, search and replace, and viewing features. This editor also allows display of text in an arbitrary number of fonts, mixing those fonts at any point in the document. This editor includes an input filter for pascal source programs that will read the file and generate a text object in which all pascal keywords are displayed in a boldface font, all variables are displayed in an italic font, and all other text is displayed in a regular font.

Perhaps more interesting is the project developing a Kanji editor using the hierarchical object model of text. This editor displays text using four character sets—7 bit ASCII, an 8-bit character set that maps codes to Japanese Katikana, an 8-bit character set that maps codes to Japanese Hirigana, and a 16-bit character set that maps codes to a 7,000 character Kanji code set. The editor allows mixing of any or all of these code sets throughout the document, changing (if necessary) from character to character.

Both of these editors make heavy use of both complex text structures and properties associated with objects in those structures. In spite of the amount of overhead this would seem to add, the performance of both is quite good on a 1mips class machine.

A type manager has also been developed for hierarchically structured text object. This manager, based on the architecture of Apollo's extensible I/O system [5] and written using our Open Systems Toolkit [6], allows access to hierarchically structured text objects through standard system calls without requiring that the program accessing the object be aware of the internal representation of the object. The manager makes use of the text management library's modelling and formatting component, and presents to the system an object that appears to be a linearly ordered stream of text. The hierarchically structured text object is transformed into a formatted version of the object, and the lines of this formatted version are returned to the

accessing program as streams of characters. It is through this manager that programs such as compilers and simple print spoolers access text objects.

More sophisticated printing is accomplished by generating a Postscript² file from the hierarchical text object. Such a file is generated by replacing the rendering component used to generate screen output by a rendering component that generates a Postscript file. Both renderers use the same representation to generate their output, namely that generated by the formatting component of the text management library.

Future Directions

The next steps we plan to take in developing the text management library center on two separate areas. The first of these involves an extension in the model of the hierarchical object structure used to model text. The second revolves around issues concerning the construction of a user interface that allows full access to all of the powers of the toolkit.

The extension to the hierarchical object structure concerns the interpretation of terminal nodes in the structure tree. Currently, such nodes are required to point to an area within the character heap. Because of this requirement, text objects currently cannot contain graphic objects. However, extending the model to allow for such objects appears to be relatively straightforward.

Such an extension would begin by introducing a property within the model of "being a graphic object." Any terminal node that had this property would also have to have a formatting property specifying a rectangular area in which the object was to be contained on a conceptual page, and a rendering property pointing to a routine that would actually render the object. The referent of the node would then be an area of data (perhaps contained in a separate file) that would be used by the rendering routine to actually paint the graphic object on the output device. The treatment of such objects by the text management library would be to reserve an area on the conceptual page large enough to contain the graphic, and to call on the rendering routine during production of the conceptual page on the output device. Such an approach avoids the need for the text management library to contain a set of graphics primitives in either the formatting or rendering component of the library. Instead, the library simply assumes that some such set of primitives will be provided to it by the application when the need arises.

The problem of finding a model for a user interface to a toolkit based on hierarchically structured text is both more difficult and, strictly speaking, outside the scope of the development of the toolkit. However, this problem must be considered if the text management library is to be fully used. Basically, the problem is as follows. The power of the toolkit comes from the representation of text as a structured tree of objects that may have properties associated with them. But the view of the object presented to the user is some version of the object in a two dimensional form, after it has been passed through the formatter and renderer. How and whether to present the user with a representation of the underlying text model, which has an object hierarchy of arbitrary depth and that can associate properties with objects at any level, is a problem we have only begun to grapple with.

Conclusion

The hierarchical object model of text is based on a separation of the representations of the characters that make up text, the structure that organizes those characters, and the properties that contain information about the text. By making this separation, we have been able to simplify the context needed to interpret a code as a character within a text object. We have also

been able to both expand the set of operations available on the text object and simplify the implementation of those operations. Most importantly, by using this model we have been able to avoid having to choose between allowing a totally general set of information to be stored as part of the text object and being able to manipulate, format, and present the text at a speed acceptable for interactive applications.

Acknowledgements

I would like to thank the members of the User Environment group at Apollo for their criticisms, comments, and suggestions on both the design of the text management library and this paper. Special thanks go to Jim Hamilton, Tom Greene, and John Yates who all contributed significant ideas during the first stages of design of the hierarchical object model of text, and to Jim Haungs for pointing out alternatives along the way. I would also like to extend special thanks to my editor, Steve Marchesano, for his aid in the production of this paper.

Notes

1. Meyrowitz and van Dam, p. 339.
2. Postscript is a trademark of Adobe Systems Incorporated.

References

1. Gutknecht, J. **Concepts of the Text Editor Lara**. *Communications of the ACM*, 28, 9 (1985), 942-960.
2. Lampson, B.W. **Bravo manual**. Alto User's Handbook, Xerox Corporation, Palo Alto, Calif., 1978.
3. Meyrowitz, N. and van Dam, A. **Interactive Editing Systems, Part 1 and 2**. *ACM Computing Surveys*, 14, 3 (1982), 321-417.
4. Nelson, T.H. **Getting it out of our system**, in *Information Retrieval: A Critical Review*, G. Schechter (ed.), Thompson Book Co., Washington, D.C., 1967, 191-210.
5. Rees, J., Shienbrood, E., Levine, P. **An Extensible I/O System**. 1986 Summer USENIX Technical Conference.
6. **Using the Open Systems Toolkit to Extend the Streams Facility**. Apollo Computer Inc., Chelmsford, Ma. 1986.

SMScript: AN INTERPRETOR FOR THE POSTSCRIPT* LANGUAGE UNDER UNIX.

Bruno BORGHI, Stéphane QUEREL, Daniel de RAUGLAUDRE

GIPSI-SM90¹
c/o INRIA
BP 105
78153 LE CHESNAY CEDEX
FRANCE

---mcvax!inria!gipsy!borghi
---mcvax!inria!gipsy!stephane
---mcvax!inria!gipsy!daniel

Abstract: In this article, we present an implementation of an interpreter for the PostScript language running on the french workstation SM90 under Unix System V. In the first part, we explain the reasons that led us to implement PostScript. Then we expose some details of implementation. Finally we describe the new features offered to the graphic and document production programs, and the PostScript environnement that can be built around this interpreter.

1. A Printing Server for the SM90

The SM90 workstation uses a multimicroprocessor architecture, developed by CNET (Centre National d'Etudes des Télécommunications), based upon 680x0 CPUs and runs a version of Unix System V called SMX V.1. Modularity of the hardware allows to finely tune the workstation to the user requirements. Bull markets the SM90 workstation under the

* PostScript is a trademark of Adobe Systems Incorporated.

¹ GIPSI-SM90 a été financé pour partie par le Ministère de la Recherche et de la Technologie, sous les contrats nos 83-B1032, 84-E0651 et 85-B0524.

name: SPS7.

GIPSI-SM90 develops specific components needed for a scientific workstation use of the architecture. To meet the requirements of the scientific community for which this workstation is designed, we are adding a full document production capability.

Constraints for such a use are known: scientists and engineers need powerful personal workstations with a high quality document processing and printing service, although this one is only used from time to time. This leads either to offer a big printer server connected to a network of workstations or to divert a small part of the CPU power of the workstation to drive a smaller printing engine attached to it.

GIPSI chose the second solution. This involved the choice of low cost printer and hardware interface. The simplicity of the hardware and the need of excellent document quality impose to offer a powerful software interface that allows complete freedom in document creation and full use of the printer qualities.

1.1. Hardware

A well-known low-cost laser printing engine (Canon LBP-CX) is connected by means of a video-interface board to the bus of a processor. The hardware interface is very simple. Basically it does the transfer of 1 bit pixels from the processor memory to the printing device. The CPU builds a bitmap image of each page to be printed in its own memory and then transfers it pixel by pixel to the printer. This way, dedicated hardware is reduced to the minimum and the CPU which is in charge of managing the printer, remains available to run ordinary programs, except at the very printing time when the transfer must satisfy real time constraints.

The characteristics of the printer (300 dots per inch, 8 pages per minute) allow good quality graphic and typography, at a reasonable cost; and it is therefore usable as an individual printer connected to the workstation, possibly shared on a network by a reduced number of nearby workstations.

1.2. Software

Such a minimum hardware interface requires a good software interface.

Any program can build an image and print it, but an intelligent driver which hides the internal mechanisms and the complex bit manipulations is handier. Such a driver should be powerful enough to provide texts with a number of pretty fonts, graphics, raster images, freedom in orientation and precision in the location of these various components on a page.

If the interface between this intelligent driver and the rest of the world is a file format, the driver can be seen as a filter by any program. Furthermore, it should present several important features:

- to allow easy communications in a heterogeneous network, to enable the installation of a remote printing server: we want it to be able to work later on bigger printing server shared over a whole network of different machines;
- to be hardware independent: application programs do not have to know the physical characteristics of the printer, except when explicitly wanted. Then a document may virtually feed any device;
- to be compatible with an existing standard to allow a good portability for computer aided publishing software.

At the time we started the study, two printer file formats were answering these requirements: Adobe's PostScript and Xerox's Interpress. These two formats are in the same family, and we decided to implement PostScript for various reasons:

- every Interpress operator has its equivalent in PostScript or may be easily emulated;
- graphical capabilities in PostScript were more powerful than in Interpress².
- PostScript uses only US ASCII as input. This is important when developing applications, writing and debugging PostScript programs. In addition transmissions on any device such as magnetic tapes, asynchronous lines, networks are straightforward
- There is already one implementation of PostScript available. Apple had marketed its LaserWriter, and we could consider it as a reference and a goal. It is always helpful to be able to verify the effects of an operator when the Reference manual is obscure, or incomplete.

The main objection that could be raised is that PostScript has no printing protocol functions. This could be overhauled by using document preprocessing with standard comments included in the document, according to Adobe's recommendations.

1.3. PostScript and Unix

Knowing that we want a low cost printer with a simple hardware interface and a powerful PostScript interface to drive the printer, the question is now how to organise the communication between the PostScript interpreter and other Unix processes.

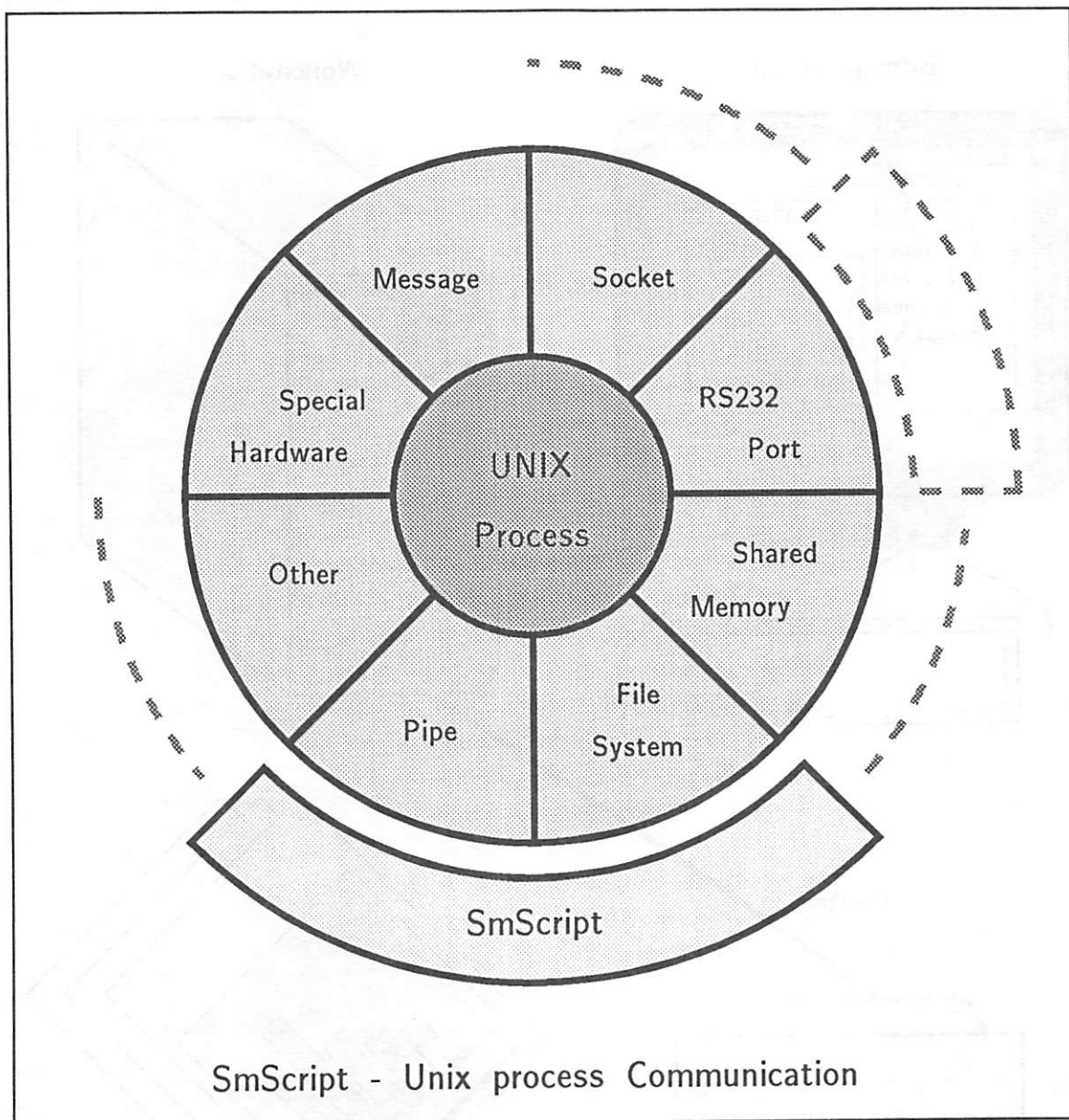
We could have chosen, as Apple did, to implement PostScript on a special board inside the printer and to offer an access to it via an RS-232 port, but this severely limits the communication between PostScript and the Unix processes.

We decided to have PostScript as a standard Unix process. This yields some advantages:

- one may use PostScript as the powerful general language;
- there is a real file system, and then no limitation in storage space for PostScript sources and character fonts;
- we can take advantage of any hardware enhancement of the machine such as floating-point facilities and 68020 CPU to reduce execution time, or bigger memories to drive printer engines with a better resolution than 300 dots per inch;
- it is extremely simple to access various kinds of raster devices through the same PostScript interface, due to the fact that PostScript is device-independent: proof-reading is possible on a bitmap screen or in a window system, without producing paper;
- PostScript is a good candidate to become a standard exchange format between applications; then, some applications should interpret PostScript sources some way.
- Though actually limited to pipes and files, communication between PostScript and other processes might use the various kinds of interprocess communication provided by Unix system V (pipe, shared memory, messages, sockets³).

² At that time we were comparing PostScript and Interpress 1.0. With the second version of Interpress, both languages are now of the same graphical level.

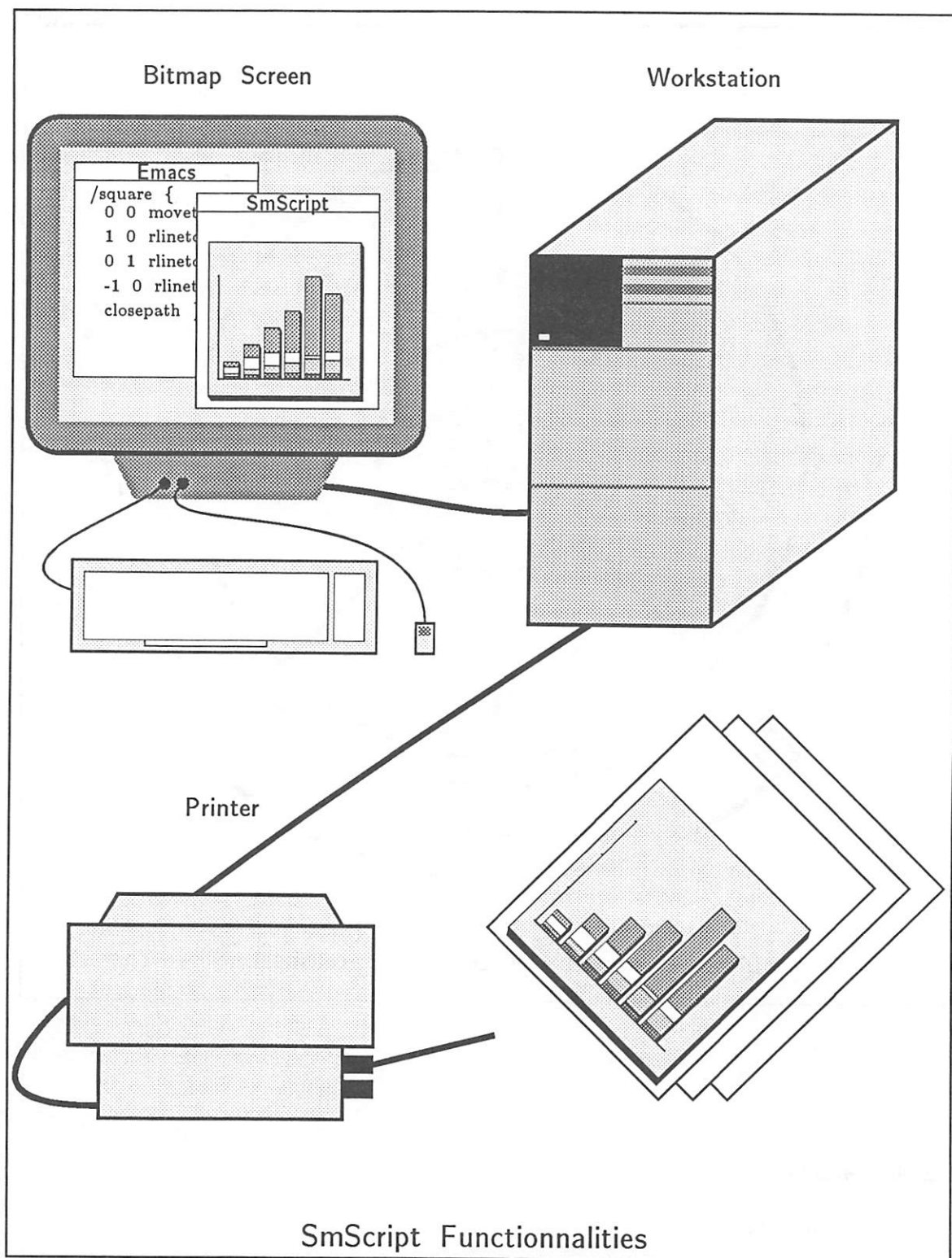
³ SMX V.1, which is System V based, offers some 4.2BSD facilities such as socket mechanism and TCP/IP network protocol.



2. Implementation

2.1. Functionalities

When we started the implementation of SmScript, we first cut down to a coherent subset of the PostScript operators. We then got familiar with the objects and the concepts, and achieved an important work on basic algorithms. From this point, we are growing to reach a full PostScript functionality.



The current subset available through SmScript is the following:

- output printed on Canon laser printer, Numelec bitmap screen, and inside a window of the SMX window system;
- most of the general (non-graphic) operators, needed to allow good programming facilities and ease in use of the language;
- use of bitmap and contour fonts;
- most of the graphic operators (Operations currently not supported are the complex clipping mechanisms and the screen management operators);
- bitmap images without transformations.

2.2. Software Architecture

The software is structured in three levels:

- *basic machines* which manipulate the data structures at the lowest level;
- *operators* which implement the native PostScript operators;
- *interpreter management modules* which perform an overall sequencing.

All the modules are written in C language or Lex and Yacc. The basic rasterops are taken from the standard SMX library: *libraster*, and are written in assembly language. The system-dependant functions are very few, and easy to isolate. This allows a good portability. The major non-portability is the use of the rasterops library, although this portability is insured for the machines with a 68000 family based CPU.

The implementation itself is independant of the printer: whatever the printing device is, the task of the interpreter is to build a bitmap image, which the hardware is in charge of depositing on the final device. Thus it is an easy task to add new printing devices, as far as it can be considered some way or another as a memory⁴.

The main choices for the implementation are related to the data structures (objects, stacks, dictionaries) and the way the modules communicate and handle errors. But we always tried to manipulate the structures as abstract types. We can thus change the data structures quite easily. For example we recently changed our stack implementation from lists to arrays with a very few changes in the code. As far as we didn't know which structure would be well suited to the problem we implement it with the procedures to manipulate it, so that we only need to change a small number of functions when changing the structures.

2.3. Graphic Processing

We developed algorithms for stroke drawing, polyline filling according to the non-zero winding rule, and region clipping. As we need a good precision, and for reasons of developing time we used floating point reals for the calculations related to the paths. The filling module uses an integer algorithm for efficiency. Our rasterops library is, in its actual version, a little bit too coarse, and our algorithms have to do some low level job especially for region filling. We are working to provide a new library, with extended primitives, which will discharge the interpreter of inefficient low level raster manipulations.

⁴ This is true for bitmap screens, with or without window systems and laser printers in raw mode such as the Canon LBP-CX printer.

The choice of floating point reals may not seem to be optimal for execution time, but the SM90 has a powerful array-processor board and 68020 cpus. Therefore the performances of the interpreter can be enhanced much more than with complex software optimisations. The graphic part of the interpreter being almost complete, we are now refining the algorithms for a faster execution time.

2.4. Character Fonts

SmScript has to use any kind of font (bitmap, vector, cubic spline). We started with bitmap fonts. Our font format was poor and not suited to manipulate and archive contour fonts.

Thus we had to design a new font format: GAFF (General Application Font Format). We build it versatile, self-descriptive and open. It is usable by various kind of software: typesetting, interactive graphic software, printing applications.

It is inspired by the PostScript font dictionaries architecture, the TEX font format, and our previous format. It handles both high level informations on the characters, which are currently needed by a text formater (widths of characters, kerning, ligatures, font name, point size, etc...) and low level informations needed by the imaging process (bitmaps to show, vectors and splines to interpolate, etc...). The design of the format allows shortcuts for processes which only need a quick access to low level information. It is also designed to be modular in order to hide informations for processes which do not need or are not allowed to access them. It permits also data encryption for security.

We are developing a *character server* to handle this format and insure that existing programs could still run, and to offer access to contour fonts to all programs without the heavy management of these fonts. This server is in charge of managing the fonts currently available in the system or over the network, and offers a unique interface either to the various kinds of contour fonts or to the bitmap fonts. The server implements a caching mechanism which optimise the use the characters by virtually sharing a character cache among the client processes of the server. It is a nice solution to the problem of using contour fonts inside a program, and it allows the powerful PostScript character management mechanism to be accessible by any application.

3. Around SmScript.

When we started SmScript, our goal was to implement a powerful printer driver. Even though SmScript has not yet reach the full PostScript functionality, it is already in good working order and greatly enhances the printing capabilities⁵ of the SM90.

Now our primary goal has evolved. We are studying the issue of SmScript as part of a complete document production package where PostScript could be used not only as a printing format but also as an exchange format between applications in a graphic environment. For example a way to implement a *cut-and-paste* function between a graphics editor and a text processor, in a window system could be:

- a set of textual or graphic applications (GKS, troff, Image processing programs, etc...) generate graphic elements as PostScript files;

⁵ The pictures of this documents were written in PostScript and printed on a Canon LBP-CX printer using SmScript.

conventional timesharing systems. The secondary benefits of TRFS between disk-based machines were also recognized from the outset.

** Transparency of applications*

No modifications to any applications or utility programs should be necessary. Moreover, since many of our customers have obtained software in binary form from independent vendors, we cannot even require that programs be recompiled or relinked. This in turn means that library routines cannot be changed either, nor can new system calls be introduced. All operations that can be performed on a local object should behave just the same on a remote object.

** Ease of use*

Specification of a remote object should be trivial. Other than that, users should not need to concern themselves with TRFS. Similarly, TRFS should not be an unwieldy burden to the system administrator.

** Performance*

TRFS must not result in noticeably slower performance.

3. Usage

3.1 The User's Perspective

Under TRFS, a remote pathname is a string of the form `/@machine_name/remote_pathname`, where *remote_pathname* is an arbitrary pathname relative to the root directory of the remote machine corresponding to *machine_name*. A remote pathname can be specified wherever a conventional pathname would normally be used. For example, the command

```
cat /@alice/etc/passwd
```

will cause the contents of the file `/etc/passwd` on the machine known as `alice` to be displayed. Any program that uses pathnames can access remote files, such as

```
vi /@alice/usr/sys/trfs/*.c
```

to edit a set of remote files with a popular text editor. Notice that meta characters get expanded in the normal way by whatever shell is in use.

Since remote pathnames can become tedious to specify (just like conventional absolute pathnames), the current directory can be changed to a remote directory, allowing subsequent use of relative pathnames to access remote objects:

```
cd /@alice/usr/sys/trfs
pr ../h/*.h *.c | lpr
```

Remote devices can be specified in the same way. For instance,

```
tar xf /@alice/dev/rmt0
```

will extract all the files from a tar tape mounted on a remote tape drive.

For diskless systems, the root directory is always the root of the disk-based server machine. All of the diskless nodes of a given server share the same entire directory tree. It is not necessary for a user on a diskless node to ever specify a remote pathname, except to access objects on yet another disk-based remote machine.

TRFS only supports access to remote objects found in the UNIX hierarchical file system. It does not deal with other remote functions such as remote login or remote execution. It was decided that the existing 4.2 & 4.3 BSD facilities were quite adequate to meet these needs.

3.2 Administration

It is possible to access remote objects in a completely transparent way without using remote pathnames, through the use of *symbolic links*. Under standard 4.2 or 4.3 BSD UNIX, a user can create a node in the directory tree which simply points to another file. Unlike a hard link, a symbolic link contains an arbitrary pathname string, allowing it to refer to a file on another file system (or even a nonexistent file). The target pathname of a symbolic link can be either absolute, or relative to the location of the symbolic link itself, depending on whether or not the first character of the target pathname is a slash.

By creating symbolic links whose targets are remote pathnames, a system administrator (or any user for that matter) can cause conventional pathnames to refer to remote objects. For example, the symbolic link

```
/lib/libc.a → /@alice/lib/libc.a
```

will cause all references to the file `/lib/libc.a` to actually refer to the file of the same name on the remote machine known as `alice`. This could greatly simplify the task of maintaining this library, making it unnecessary to track down and modify a plethora of copies of this file whenever it is updated. Note that programs which "know" the specific pathname `/lib/libc.a` (such as the C compiler) will be unaware that the file being accessed is actually on a remote machine.

Remote devices can also be the targets of symbolic links. Recalling the previous `tar` example, if the system administrator creates the symbolic link

```
/dev/rmt0 → /@alice/dev/rmt0
```

then `tar` can be invoked without the explicit remote tape device specification. The command

```
tar x
```

will extract the files from the default tape device (`/dev/rmt0`), and the unmodified `tar` program will be oblivious to the fact that a remote tape device is actually being used. (The antisocial consequences that can result when two or more users attempt to access the same tape drive at the same time are outside the scope of TRFS. In any case, this situation is no different than with a multi-user timesharing system.)

Since symbolic links can also refer to directories as well as regular files or devices, it is easy to share entire directory subtrees. The symbolic link

```
/usr → /@alice/usr
```

will cause the entire `/usr` subtree on the remote machine `alice` to be shared by the machine

containing the symbolic link.

The use of symbolic links in this way provides a capability similar to the "remote mount" facility used in other distributed file system implementations. We decided that the combination of remote pathnames and symbolic links is sufficiently powerful to render remote mounting unnecessary. With remote pathnames, any file or device on any TRFS machine on the network is accessible (subject of course to the constraints of the normal UNIX protection mechanism). With remote mounting, if the file you want to access is not contained in some remote subtree which has been explicitly mounted on your machine, you're out of luck unless you can find a super-user to perform the necessary remote mount incantation. Where total pathname transparency is required, TRFS supports symbolic links to remote objects.

As mentioned previously, users on diskless workstations do not generally need to specify remote pathnames, since the root directory of a diskless node is established at boot time as the root directory of the associated disk-based server node. This implies that the entire directory hierarchy of the server and every file therein will be shared by each of its diskless nodes. While a highly shared file system such as this is generally beneficial, there are some particular files and directories which cause difficulties.

Consider for a moment the file `/etc/ttys`. This file contains information which specifies which terminal ports should have logins enabled when the system is running multi-user. This information will almost always be different for a server machine, which is often configured with many serial ports and no graphics, and a single user diskless workstation, which usually has a graphics subsystem and no additional serial ports. For this reason, it is not suitable for this file to be shared.

Temporary files present a similar problem when multiple machines share the root file system. Various programs create temporary files in the `/tmp` directory, deriving unique names from the running program's unique process id. Unfortunately, a process id is only unique on a given processor. When two or more machines share a common `/tmp` directory, there is a very real possibility of temporary filename collisions.

In order to deal with shared files and directories that really need to *not* be shared, we introduced *private links*. Private links are normal symbolic links whose target pathnames contain special meta sequences. These meta sequences get replaced when the link is followed. The replacement string is dependent upon the name of the machine from which the request originated.

The basic private link meta sequence is `"$HOST"`. Whenever this string is encountered when following a symbolic link, it is replaced by the name of the machine running the process from which the request originated. Note that this is a literal 5-character string of which the first character is a '\$', not to be confused with a shell or environment variable. The `$HOST` private link is actually not used very much. More common is the `$RHOST` private link. The `"$RHOST"` meta sequence gets replaced by the name of the the originating host, unless the process is running on the machine containing the link, in which case the replacement string is the null string. Consider, for example, a server named `larry` and two diskless nodes called `moe` and `curly`. If the following symbolic link and files are created

```
/etc/ttys → /etc/.ttys.$RHOST
/etc/.ttys.
/etc/.ttys.moe
/etc/.ttys.curly
```

then whenever a program running on `moe` refers to `/etc/ttys` it will actually access the file `/etc/.ttys.moe`. When `larry` refers to `/etc/ttys` it will get `/etc/.ttys..` Similarly, the symbolic link and directories

```
/tmp → /.tmp.$RHOST
/.tmp.
/.tmp.moe
/.tmp.curly
```

is all that is needed to provide each machine with its own private `/tmp` directory.

By using `$RHOST` private links instead of `$HOST`, we avoid imbedding the host name of the server in various and sundry places all over the disk. This prevents problems when changing the host name of the system, and also allows access to the appropriate files by the local machine when running single-user, before the host name has been set. The leading `'.'` in the target file names is simply a convention used to hide things from the uninitiated.

Any of the private files can actually be accessed from any of the machines, either by specifying the target file name explicitly (for instance, `/etc/.ttys.curly`), or by using a remote pathname (`/@curly/etc/ttys`). When a machine name is specified in a remote pathname, the private link is expanded as if the request originated from the specified machine. This allows a user on `moe`, for instance, to explicitly copy a file into `curly`'s `/tmp` directory without having it end up in `/.tmp.moe`.

Remote pathnames and private links are the only visible changes caused by TRFS. Remote pathnames promote sharing of files and devices among multiple independent machines. Symbolic links can be used to help make this sharing more transparent by hiding the remote pathnames.

Private links are only needed to help support diskless nodes. In effect, they temper the overly-enthusiastic sharing of files and directories among machines which have a common root directory.

4. Implementation

4.1 The Protocol Interface

The interface to the underlying protocol used by TRFS was derived from the V Kernel[1], developed at Stanford (although our UNIX implementation of this protocol is of our own design). The interface defines an asymmetrical client-server model of interaction, in which the client process sends a message to the remote server process and then waits for a reply. The server receives the message, processes the request, sends back a reply, then waits for another request message. While processing the request, the server may optionally transfer some data to or from the client.

The following are the basic primitives defined by the protocol:

`message, rpid ← Send(message, rpid)`

Causes a fixed-length message (64 bytes) to be sent to the process specified by `rpid`.

The message can optionally contain one or two *segments*, each consisting of an address, a length, an access mode (read and/or write), and an address space qualifier (user, kernel, or physical). Upon return, the message has been overwritten by the reply message. This is the only protocol primitive used by the client process.

`message, rpid ← Receive(rpid)`

Waits until a message arrives from the specified process (or anybody if `rpid` is zero). If an appropriate message has already been queued up then return is immediate.

`Reply(message, rpid)`

Causes a fixed-length reply message to be sent back to the specified process, who should be blocked in `Send`. Return is immediate.

`CopyTo(rpid, srcaddr, dstaddr, len, srcspace, dstspace)`

Copies `len` bytes of data from `srcaddr` in the caller's address space specified by `srcspace` to `dstaddr` in `rpid`'s address space specified by `dstspace`. The remote process must be blocked in `Send`.

`CopyFrom(rpid, srcaddr, dstaddr, len, srcspace, dstspace)`

Copies `len` bytes of data from `srcaddr` in `rpid`'s address space specified by `srcspace` to `dstaddr` in the caller's address space specified by `dstspace`. The remote process must be blocked in `Send`.

4.2 The Higher Levels

The above routines provide an ideal interface upon which to build the higher levels of TRFS. A client process is simply any process which is attempting to access a remote object. Its corresponding server process is termed a *cousin*. A cousin is a normal process in all regards except that it has no user address space. Each cousin provides services to a single client process. A new cousin is created on a remote machine when a process tries to access an object on that machine. A cousin replicates when its client process forks, and terminates when its client exits. Because there is a unique cousin (per machine) associated with each client, and because of the synchronous nature of the protocol interface, each cousin process functions as a logical extension of the corresponding client process.

As a process, each cousin maintains a complete process state (with the exception of user address space). In particular, a cousin has some number of open files and a current directory. Each client also has a complete process state, but in addition, the client keeps track of the mapping between those of its file descriptors which are marked remote and its cousin's file descriptors. The client also remembers on which machine (and hence by which cousin) the current directory is located.

When in the course of executing a system call a process determines that remote service is required, either because a pathname is on a remote machine (remote pathname, remote current directory or remote root directory) or because a file descriptor corresponds to a remote object, the process builds an appropriate request message containing all the information that the cousin will need to service the request. Pathnames are passed not in the message itself (which is of fixed length), but rather in a segment. Other segments may also be set up if it is expected that the cousin will require access to the client's address space to service the request. The client calls `Send` and blocks.

For most operations, the cousin process receives the message, determines the nature of the

request, sets up the parameters so that it appears that the system call originated locally, and calls the appropriate kernel system call handler. Things proceed as if the cousin was a normal process until it tries to access user space. There are only a handful of routines in the kernel which are used to access user memory (`copyin`, `copyout`, `fubyte`, ...), and they have all been modified to check whether the calling process is a cousin. If so, then either `CopyTo` or `CopyFrom` (see above) is invoked instead. Thus the data gets transferred between the kernel space on the cousin machine and the user's address space on the client machine, just like it should. At this low level the protocol doesn't know (or care) whether the system call handler is reading in a pathname, or writing out the contents of a disk buffer, or whatever.

When the operation is complete, control is returned from the system call handler to the cousin request-processing code. Any appropriate return values are stuffed into a reply message, `Reply` is called, and the cousin goes back to waiting for another request. The client, upon receiving the reply, updates its state-mapping information if necessary, and then completes execution of the system call.

4.3 The Lower Levels

The protocol itself is the implementation of the interface described above. It is designed to run reliably using a packet-switched communications medium such as ethernet. In particular, the ethernet level only provides unreliable datagram communication in which packets can be dropped, duplicated or reordered. Therefore this protocol performs timeout and retransmission, redundant packet filtering, and packet resequencing.

Figure 1 shows the packet transfers involved in a `Send-Receive-Reply` transaction. There is not much room for performance improvement with this simple operation.

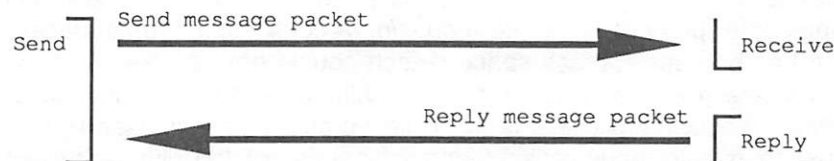


Figure 1: Basic `Send-Receive-Reply` transaction

Figure 2 denotes the packet transfers in a `Send-Receive-CopyFrom-Reply` transaction. The `CopyFrom` request sent from the server back to the client contains the address space qualifier and source address from which the data is to be sent, as well as the desired length in bytes. The data is transmitted by the client in a *burst*, consisting of one or more data packets. Each data packet is labeled with the total number of packets in the burst as well as the index of this particular packet. If any of the data packets go astray, or if the total amount of data contained in the burst is less than the amount requested, then the server will issue one or more new `CopyFrom` requests, specifying only the regions which were not successfully received.

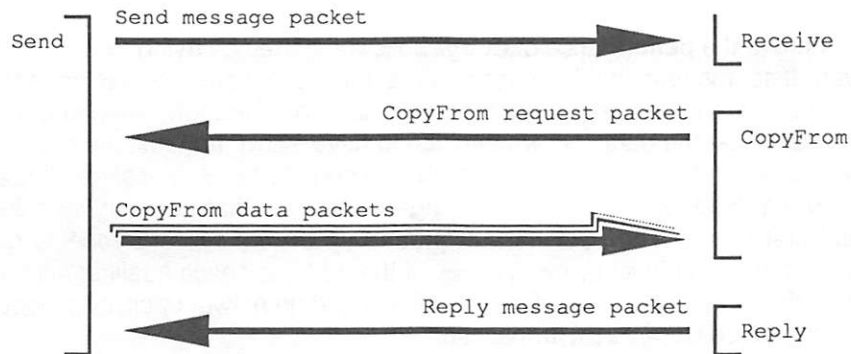


Figure 2: Basic transaction with CopyFrom operation

The number of packets transmitted in a burst is up to the sender of the burst, and is based on the amount of network transmit buffer memory available and the perceived ability of the recipient to successfully receive all the packets in the burst. Each machine maintains a *maximum burst length* for every other machine with which it communicates. If the remote machine reissues requests for data which has just been successfully transmitted, then it is assumed that he is having difficulty receiving all of the packets in the burst and the maximum burst length associated with that machine is quickly reduced. After a while the maximum burst length is slowly nudged back up, to ensure that performance is not permanently degraded because of a transient condition.

The maximum burst length thus functions as a throttle control, to help keep a slower machine (such as a 68010) from choking every time it receives data from a faster machine (such as a 68020).

Figure 3 denotes the transfers which occur in a Send-Receive-CopyTo-Reply transaction. This is very similar to the CopyFrom case, except that, instead of a single request packet acknowledged by multiple data packets, the CopyTo request takes the form of a multi-packet burst acknowledged by a single packet. Again, each packet in the burst is marked with the total number of packets in the burst, allowing the recipient of the data to determine when all of the data has been received. If no acknowledgement is received within a timeout period then a single-packet burst with the same sequence number is sent. The client will immediately respond with an acknowledgement, indicating which packets of the original burst have been successfully received and processed. The server can then initiate one or more new CopyTo operations to transfer the remaining data.

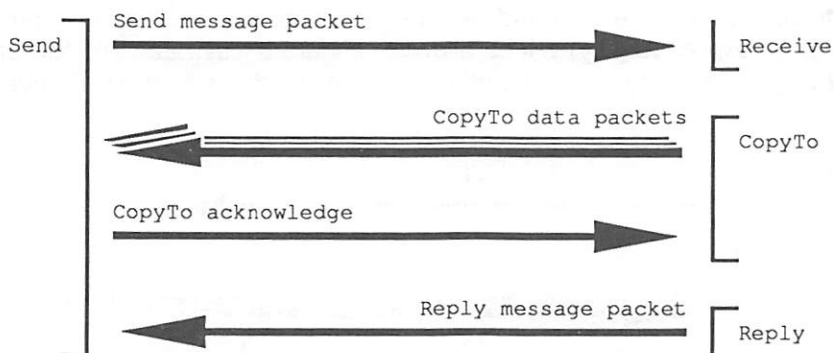


Figure 3: Basic transaction with CopyTo operation

In order to improve the performance of `CopyFrom`, we started by trying to anticipate the behavior of the server. It seems reasonable to assume that if the client process has gone to the trouble to set up a segment for the server to read, the server is in fact probably going to issue a `CopyFrom` request to actually get the data. So we decided to have `Send` transmit the data contained in its readable segments at the same time it transmits the original `Send` message. It appends the first of the data to the fixed-length request message and sends that in a single packet, then sends up to a full burst more. The packets are queued up on the server's receive queue, and are checked when the server invokes `CopyFrom`. If the address space qualifier and source address region match, then the data is immediately available and no network activity is necessary. Figure 4 shows the improved `CopyFrom` transaction.

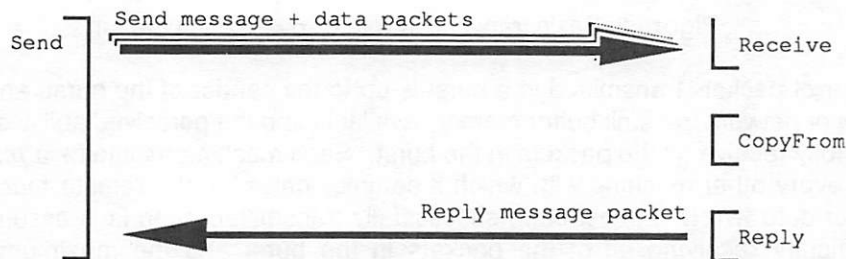


Figure 4: Improved `CopyFrom` transaction

This optimization is especially effective for any system call which takes a pathname as an argument, as well as for the `write` system call.

After careful examination of `CopyTo`, we determined that it would be worthwhile to send the `Reply` message along with the last packet of the last `CopyTo` burst. To allow this to occur, we added the following routine to the protocol interface.

`ReplyOnCopy(message, rpid, addr, xs)`

Stores the specified reply message for subsequent transmittal to `rpid` if we do a successful copy operation to or from `addr`. The `xs` argument indicates the address space associated with `addr` and the direction of the triggering copy (read or write).

This routine is used in anticipation of a successful full length copy. The `message` should contain exactly that which would be sent if no error conditions occurs. The `addr` generally corresponds to the last byte of a segment. This routine cannot be invoked unless all the return parameters corresponding to a successful full-length copy are known in advance. After the specified byte has been accessed and the reply message has been sent, no further copy operations (nor an explicit `Reply`) will be allowed. Figure 5 illustrates this sort of transaction. Note that the `Reply` message is delivered in the same packet with the last chunk of `CopyTo` data.

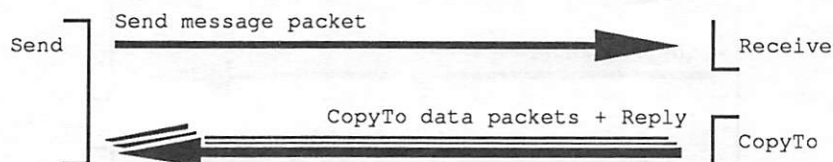


Figure 5: Improved `CopyTo` operation

`ReplyOnCopy` was implemented mostly to improve the performance of the `CopyTo` operation, (for the benefit of the `read` and `stat` system calls) but it can be used to advantage for `CopyFrom` as well. In this situation, although the sequence of packets will not be any different, the `Reply` packet will be sent as soon as the server has determined that all the expected data packets have been received and queued up, even before they have actually been processed. This results in significantly better overlap of execution between the two processors.

These optimizations can be used in conjunction with each other. For example, in performing the `stat` system call, the pathname is transferred in the same packet as the `send` message, and the `stat` structure is returned with the `Reply` message.

4.4 Controller Architecture

The VME bus driver and the ethernet driver provide the same interface to the rest of the UNIX kernel. In addition to providing a standard 4.2 / 4.3 network driver interface, they also support a faster interface to support the new protocols. The new interfaces, as well as the ethernet controller itself, were designed to help the protocols run with no unnecessary copying of data.

The only extra hardware required to support the bus driver is a shared memory board, accessible to all the processors on the bus. This memory contains a number of fixed-length buffers (each slightly larger than 8 kilobytes) in a freelist, and an input queue for each processor. To transmit a packet, a process simply gets a buffer from the freelist, copies the packet into it, places the buffer on the destination processor's input queue, and interrupts the destination processor. The receive interrupt service routine unlinks the buffer from the input queue and places it on the appropriate process' receive queue, awakening the process if necessary. When the recipient process is through with the packet, it is returned to the shared memory freelist.

The ethernet controller was designed with the same sort of interface in mind. It consists of some shared memory, accessible to the host over the VME bus, as well as the ethernet interface logic and a 68010 processor to run the controller. The shared memory contains a freelist of fixed length buffers (in this case the maximum allowable ethernet packet size), an output queue and an input queue. To transmit a packet, a process gets the next buffer from the freelist, copies the packet into it, and links it on the output queue, interrupting the processor on the controller if the queue had been empty. The controller takes care of transmitting the packet over the ethernet and returning the buffer to the freelist. As with the bus driver, the ethernet driver's receive interrupt service routine unlinks each received buffer from the input queue and places it on the appropriate process' receive queue. The recipient process will return the buffer to the controller's freelist when it has finished processing it.

At the time one of the driver's transmit routines is called, the packet itself may not have yet been completely constructed. All the header information has been put together, but if there is any data to be included then the driver must get the data after obtaining a transmit buffer. Since the currently-running process is in fact the process for which the packet is being transmitted, the data can be copied in from user address space at this time.

In the case of received packets, the data is not copied out of the receive buffer during the interrupt service routine. Rather, the receive buffer itself is made available to the destination process. When the data in the buffer is finally accessed, it can be copied directly out into user address space by the receiving process itself.

Figure 6 shows the data copies that occur with the standard driver interfaces and a typical DMA-type ethernet controller. Figure 7 illustrates the copies that take place with the new driver interface and the memory-mapped ethernet controller. Figure 8 shows the same operation as

performed by the bus driver.

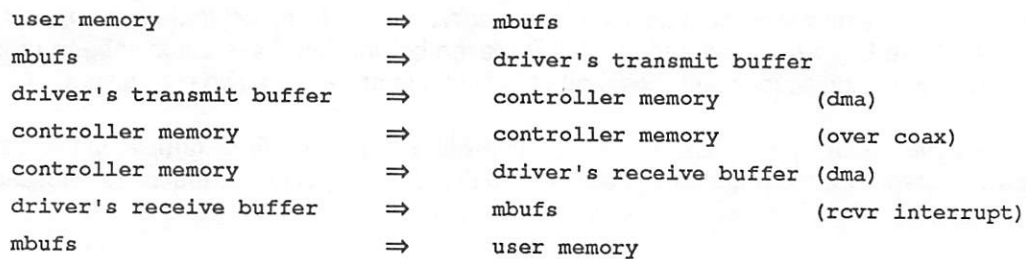


Figure 6: Data copy operations with conventional ethernet controller

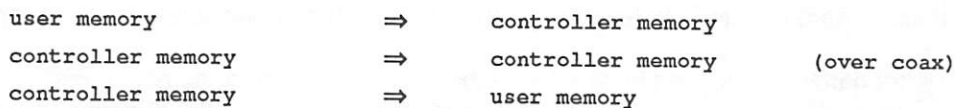


Figure 7: Data copy operations with new memory-mapped ethernet architecture



Figure 8: Data copy operations with VME bus driver

4. Performance

In order to measure the performance of TRFS, we first installed some local system calls to allow direct access to the protocols from a program running in user mode. A pair of test programs were written (a supplier and a consumer) to repeatedly send a message and copy an 8 kilobyte buffer from one user address space to another. Figure 9 shows the results.

Ethernet:	514 kilobytes per second
VME bus:	1.33 megabytes per second

Figure 9: Sustained protocol throughput, user to user, 8 kilobyte buffer length

Performance of the file system is more difficult to measure, due to the limitations of today's disk drive technology. Unless the test programs were contrived to ensure disk buffer cache hits, the disk was always the limiting factor in our tests. Figures 10 and 11 show file read and write throughput with cache hits.

Ethernet:	409 kilobytes per second
VME bus:	1.03 megabytes per second

Figure 10: File read throughput with disk buffer cache hits

Ethernet:	493 kilobytes per second
VME bus:	1.25 megabytes per second

Figure 11: File write throughput with disk buffer cache hits

4. Conclusions

The Transparent Remote File System seems to have successfully met its design goals. It provides a high degree of transparency for application programs, and is easy to use and administer. Performance is sufficient to support diskless workstations without discernable performance degradation as compared to disk-based systems.

Future work includes continued analysis of performance characteristics. Among additional optimizations being considered are file readahead by the client and local ram disks for temporary files.

References

- [1] David R. Cheriton, "The V Kernel: A Software Base for Distributed Systems," Computer Science Department, Stanford University (1984)
- [2] Dennis Richie and Ken Thompson, "The Unix Time-Sharing System," *Communications of the ACM*, pp.365-375, The Bell Telephone Laboratories (1974)

A Global Optimizer for Sun FORTRAN, C & Pascal

Vida Ghodssi, Steven S. Muchnick & Alex Wu

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043

1. Introduction

At Sun Microsystems we have recently released a new global code optimizer in our FORTRAN 77 compiler and are currently in the process of adapting it to our C and Pascal compilers. The combination of the global and peephole optimizers has resulted in measured improvements in execution speed of typical FORTRAN code of anywhere from 10 to 80%. The cost, as with any other optimizer, has been an increase in compilation time, generally by a factor of about two when the optimizer is used.

With the exception of the global optimizer, Sun's FORTRAN, C and Pascal compilers are significantly enhanced versions of the 4.2 BSD UNIX* compilers. The enhancements include better code generation, syntactic extensions to C and Pascal, full support for interlanguage calling, and window-based symbolic debugging with *dbxtool* [AdaM85]. In contrast to the other parts of the compilers, the 4.2 BSD global optimizer was discarded *in toto* and replaced with a new optimizer called *iropt* (Intermediate Representation Optimizer), which has proven to be much more effective and robust than the original 4.2 BSD optimizer. *Iropt* transforms a language- and machine-independent intermediate representation of programs and so is relatively easy to port to all of our compilers.

It is significant to note that for some simple FORTRAN benchmarks [Much86] a Sun-2 system provides essentially the same performance as a VAX-11/780 running VMS FORTRAN or a VAX-11/785 running UNIX FORTRAN, all optimized, while a Sun-3 is about four times as fast. The code quality is frequently excellent: for example, the optimized code produced for the inner loop of the Linpack benchmark [Dong86] targetted to either the Motorola MC68881 or the new Sun Floating-Point Accelerator is identical to the best possible hand-generated code.

This paper describes the global optimizer *iropt* and our much enhanced version of the *c2* peephole optimizer, the optimizations they perform, the effectiveness of the optimizers and our plans for their future development.

2. Compiler Structure

With the addition of the global optimizer, our FORTRAN compiler has the structure shown in Figure 1; the dashed path represents unoptimized compilation.

The first phase of the compilation process is the front end which scans and parses the source-language statements constituting a procedure into an intermediate language called Sun IR,

*UNIX is a trademark of AT&T Bell Laboratories.

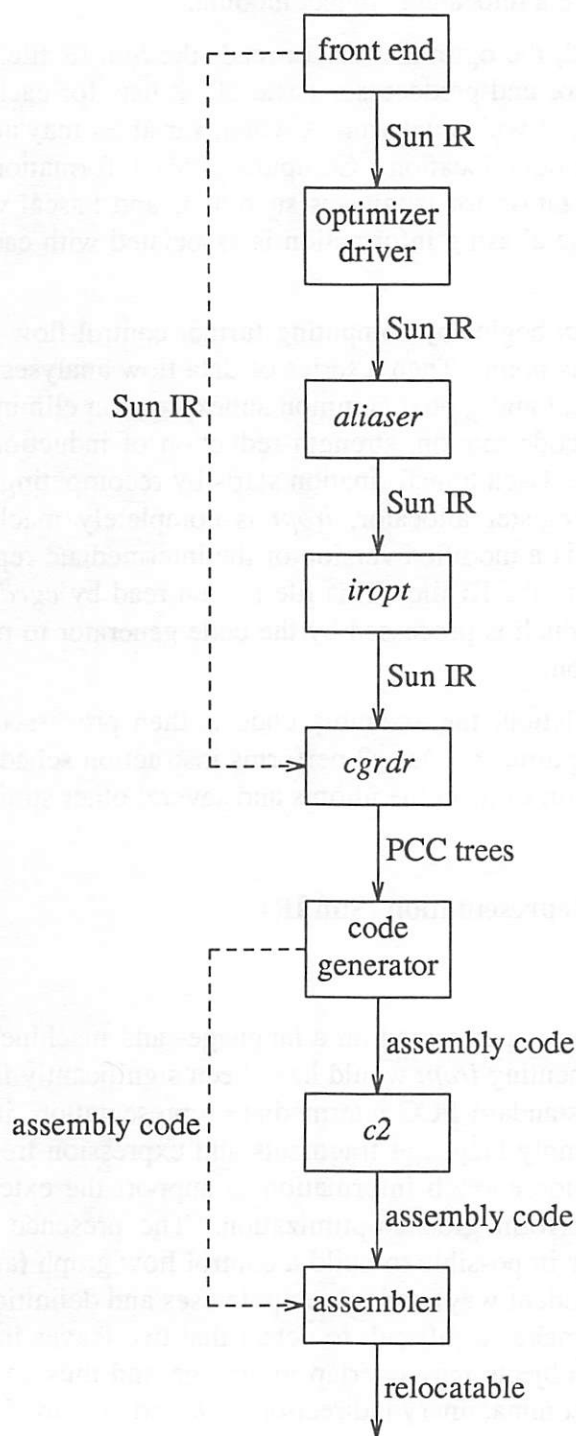


Figure 1. Structure of the Optimizing FORTRAN Compiler

which consists of a linked list of triples (which correspond in a rough way to a postorder traversal of the corresponding forest of PCC trees[Lion79]) and several kinds of auxiliary information. If optimization is not enabled, the *cgrdr* phase then translates the Sun IR form into PCC trees for processing by the code generator. The code generator, in turn, produces assembly language

which is assembled to produce a relocatable object module.

If optimization is enabled, the optimizer driver reads the Sun IR file, identifies basic blocks [AhoS86] and builds successor and predecessor basic block lists for each block. This information is then passed to the *aliaser* which determines which variables may at some point in the procedure map to the same memory location. Computing this information is essential to doing ambitious and correct optimization for languages such as C and Pascal where pointer variables may be used extensively. The aliasing information is associated with each triple for use by the global optimizer phase.

The global optimizer *iropt* begins by computing further control-flow information; for example, loops are identified at this point. Then a series of data flow analyses and transformations is applied to the procedure: local and global common subexpression elimination, local and global copy propagation, invariant code motion, strength reduction of induction-variable expressions, and global register allocation. Each transformation starts by recomputing data flow information. With the exception of the register allocator, *iropt* is completely machine-independent. The result of the transformations is a modified version of the intermediate representation of the program, which is written back to the IR file. This file is then read by *cgrdr* to generate the PCC-tree form of the procedure, which is processed by the code generator to produce assembly code, as for unoptimized compilation.

During optimized compilation, the assembly code is then processed by a much enhanced version of the *c2* peephole optimizer. Our *c2* performs instruction scheduling, register coalescing, branch chaining, utilization of machine idioms and several other straight-line code improvements.

3. The *iropt* Intermediate Representation (Sun IR)

3.1. Overview of Sun IR

Within *iropt*, optimization is performed on a language- and machine-independent representation called Sun IR. Implementing *iropt* would have been significantly faster if it could reasonably have been built on the standard PCC intermediate representation, in which a procedure is represented by a list of assembly language fragments and expression trees. Unfortunately, this representation does not provide enough information to support the extensive control and data flow analysis required to perform global optimization. The presence of assembly language instructions makes it virtually impossible to build a control flow graph (and certainly impossible to do so in a machine-independent way) and to compute uses and definitions of objects. In addition, lack of a symbol table makes it difficult to detect that two leaves in the PCC trees refer to the same object or that two objects may overlap in storage and thus be aliases for each other. Some PCC operator such as comma, unary indirection, `&&` and `||` are difficult for an optimizer to deal with.

In *iropt*'s intermediate language (Sun IR), a procedure is represented by a sequence of triples and auxiliary tables designed to remedy the shortcomings of the PCC intermediate representation. A Sun IR file consists of a header and four tables of interlinked structures describing blocks, triples, leaves and lists.

The table of leaves constitutes the symbol table for the procedure. Each constant or variable used in the procedure is described by a separate leaf structure. Variables are tagged with

(segment,offset) pairs representing their addresses and constants with their values. Separate segments are used for externals, statics, automatics, parameters, and heap variables.

A procedure's control flow is represented by a directed graph of single-entry, single-exit nodes, each of which represents a basic block. A distinguished block marks the entry point.

Computations within each basic block are represented by a list of triples of the form operand1 op operand2. The set of operators is sufficiently powerful to eliminate the need for any assembly language fragments, yet it is much simpler than the PCC operators. One of the operands of a triple may be a list of triples to accommodate operators which may require more than two operands.

List blocks are used to link structures which can be threaded in several ways. For example, the triple $a + b$ will appear on at least three lists, the lists of references for the leaves a and b and the list of triples for the block containing $a + b$.

3.2. Sun IR Operators

Triples in Sun IR are based on the 42 operators listed in Figure 2. A simple C program and the corresponding Sun IR are shown in Appendix A.

Operator	Meaning	Operator	Meaning
ADDROF	address of	LABELDEF	label definition
AND	logical and	LABELREF	label reference
ASSIGN	assignment	LE	less than or equal
CBRANCH	conditional branch	LSHIFT	left shift
COMPL	one's complement	LT	less than
CONV	convert	MINUS	binary minus
DIV	divide	MULT	multiply
ENTRYDEF	defined on proc. entry	NE	not equal
EQ	equal to	NEG	negate
EXITUSE	used at proc. exit	NOT	logical not
FCALL	function call	OR	logical or
FVAL	function value	PARAM	parameter
GE	greater than or equal	PASS	pass through as is
GOTO	go to	PLUS	binary plus
GT	greater than	REMAINDER	remainder
IFETCH	indirect fetch	REPEAT	repeat a loop body
IMPLICITDEF	implicitly defined	RSHIFT	right shift
IMPLICITUSE	implicitly used	SCALL	subroutine call
INDIRREF	indirectly referenced	STMT	pass through stmt. number
INDIRGOTO	indirect go to	SWITCH	switch
ISTORE	indirect store	XOR	exclusive or

Figure 2. Sun IR Operators

Most Sun IR operators perform the usual arithmetic, logical and branch operations, such as AND, MULT, LSHIFT and CBRANCH. Other operators serve housekeeping functions, such as helping *iropt* keep track of "hidden" definitions and uses of variables. The side effects associated with use or definition of an object are made explicit by the IMPLICITUSE and IMPLICITDEF operators.

Some operators have been introduced to disambiguate PCC operators. The LABELREF and LABELDEF operators distinguish label references from label definitions. The IFETCH and ISTORE operators distinguish different uses of PCC's U* (address indirection) operator.

Embedded assembly language fragments are avoided by using SWITCH and REPEAT operators. Code for procedure prologues and epilogues is generated by the code generator from information passed along in a header preceding the Sun IR. The ?:, && and || PCC operators have been replaced by sequences of CBRANCH, GOTO and ASSIGN triples which implement their semantics. The ++ and -- PCC operators have been replaced by their expansions into assignments and fetches. Exposing the details of the intermediate code for them allows better global optimization. The embedded assignment operator has been eliminated in a similar way.

4. The Global Optimizer *iropt*

The global optimizer *iropt* performs a series of data flow analyses and transformations, as detailed in the following sections.

4.1. Loop-Invariant Code Motion

Loop-invariant code motion finds those computations within a loop that yield the same results for each iteration of the loop and moves them out of the loop. *Iropt* computes dominator information for each basic block and uses back edges to detect and construct loops [AhoS86]. Hence it detects and optimizes both loops that result from *for* and *do* statements and those that constructed from *ifs* and *gotos*. A variable is a loop invariant if either it is not defined within the loop or is defined only once by a loop-invariant computation and only that definition reaches all the uses within the loop. A loop-invariant definition which reaches uses outside a loop is moved out of the loop only if the definition dominates all the exit blocks of the loop. A subexpression is loop-invariant if all its operands are loop-invariant. Loop-invariant computations are moved out of loops and their values are saved in a temporary variables. All uses of that value within the loop are replaced by the temporary variable.

4.2. Induction-Variable Strength Reduction

Strength reduction replaces slower operations (e.g. multiplications) by faster ones (e.g. additions, shifts). *iropt* uses the algorithm presented in [Cock77] to find the more expensive induction variable computation and replace them with addition. After the strength reduction, if any induction variable would be dead except for use in a loop closure test, the test is rewritten and the induction variable is deleted by the dead code elimination process.

4.3. Common Subexpression Elimination

Common subexpression elimination saves expression values and reuses them instead of recomputing them. To do local common subexpression elimination, *iropt* computes the local

available expression information, and then walks backward through one basic block at a time to find all maximal common subexpressions within the basic block. It deletes the redundant computations and replaces them by references to the earlier ones. After local common subexpression elimination, each basic block can have at most one computation of an expression which can reach outside uses. To do global common subexpression elimination, *iropt* computes the available expressions for each basic block in the procedure. For each use of an expression, if there is more than one computation available at some point, a temporary variable is created if there is none already for its value. All available computations of this value which are not in the same basic block as the use are stored in the temporary variable. Then the local computation is deleted and the temporary variable is used instead.

4.4. Dead Code Elimination

An expression computation is *dead* if there is no path along which the computation can reach any uses. A variable definition is *dead* if it cannot reach any uses or it can only reach computations which are used only in that definition itself. All dead computations are deleted and the reaching definition information is updated, possibly introducing more dead computations to be eliminated. Unreachable code is deleted by the peephole optimizer.

4.5. Copy Propagation

Copy operations (which assign a simple value to a variable) are of interest to copy propagation if, at run time, the source of the assignment can be referenced faster than its target. For each such copy, all local uses of the target which can be reached by this copy are replaced by the source, if the source is not redefined between the copy operation and the uses.

To do global copy propagation, *iropt* use the algorithm presented in the [AhoS86] to computes available copies for each basic block. If a copy is available at entry to a basic block, all uses of its target within that basic block are replaced by the source, if neither the source nor the target is redefined between the entry to the basic block and the uses. After the replacement all the useless definitions are deleted.

4.6. Register Allocation

All scalar objects (external variables, local variables, parameters and temporaries) that are referenced and can fit in machine registers are candidates for allocation to registers, either throughout the procedure or within a subregion of it. A global register allocator must answer two questions: First, which objects are worth putting in registers? Second, which object can share registers with others within a region of code? In addition, because *iropt*'s register allocator allocates registers to external variables, it must be able to determine where and when to store those variables back into memory.

For each candidate, *iropt* uses reaching definition information to construct bipartite directed graphs called *webs* [CouH86] which connect definitions and uses. First, *iropt* builds a web for each definition along with all the uses that definition can reach. To avoid generating code to copy values between webs, it then merges those webs that contain uses of the same resource into one. There may be one or more webs for each candidate resource. Webs are the units which are allocated to registers.

Next, the benefit of putting each web into a register is computed, based on loop nesting depth and a machine-dependent table.* The table contains an entry for each Sun IR operator which indicates the benefit of putting its operands into a register. The benefit is determined by the number of machine cycles saved if the web is allocated to a register instead of memory. The cost of restoring a register back to memory or of loading it from memory is represented by a negative number. This may happen when an external variable is allocated to a register prior to or right after a procedure call. For each loop nesting level, the benefit is multiplied by eight. The life span of a web is defined as the set of basic blocks in which the resource represented by the web must be contiguously allocated. For webs that are live only within one basic block, the life span is a sequence of triples starting at the first definition and ending with the last use. The webs are sorted by allocation benefit and then scanned in order. Webs with disjoint life spans are merged and allocated to the same register, if one is available. All references to webs which have been allocated to registers are replaced by references to the registers.

5. The Peephole Optimizer

In addition to eliminating jumps to jumps, deleting redundant loads, stores and unreachable code, inverting loops and performing other machine-dependent optimizations, our *c2* peephole optimizer does instruction scheduling and register coalescing.

5.1. Instruction Scheduling

Sun-3 systems may include an optional Floating-Point Accelerator (FPA) built around 16.67 MHz Weitek 1164 and 1165 chips. The Sun-3 CPU (a Motorola MC68020 processor) running at 16.67 MHz fetches all instructions and dispatches the FPA instructions to the FPA's instruction queue. FPA instructions can be executed in parallel with CPU instructions, as long as there are no data dependencies between the two units and the FPA instruction queue is not full. There are eight FPA shadow registers which can be read in parallel with the execution of other FPA instructions, even when the FPA instruction queue is full, as long as the registers are not busy. The FPA executes most floating-point arithmetic operations (except divisions) in four to nine cycles, faster than the CPU can send three FPA instructions to its instruction queue and in about the same amount of time required for the Sun-3 CPU to execute four register-to-register instructions or one memory-reference instruction. On the average, the FPA instruction queue is filled up only when the CPU fetches and sends seven consecutive instructions to the FPA.

For each basic block, *c2* uses resource definition and use information to build an instruction dependency DAG [GibM86] and then rearranges the instructions to maximize the throughput of both the CPU and the FPA. *c2* divides the instructions into three classes: FPA instructions, MC68020 register-to-register instructions, and MC68020 memory-reference instructions. It models the FPA instruction queue and selects instructions from among the available candidates by the following algorithm, if there is more than one candidate at a time:

If the FPA instruction queue is not full and there is at least one FPA instruction which will not cause a register interlock with already scheduled instructions, then choose one such

*For architectures with multiple register types, such as the M68000 family, it assigns different values to allocation to registers of different types.

instruction. Otherwise, choose an MC68020 instruction, if there is one.

Always choose MC68020 memory-reference instructions rather than register-to-register instructions, if possible.

Among instructions in the same class, choose one that has the maximal number of FPA instructions dependent on it.

5.2. Register Coalescing

Because we partition register allocation between the global optimizer and the code generator and because the code generator uses simple pattern matching to generate code, it generates many superfluous register-to-register moves. *c2* scans all the instructions within a basic block and uses the registers' live, use and dead information to eliminate unnecessary moves whenever possible. For instance, given the C statement

```
int1 = int2 + int3
```

assuming the register allocator puts all three variables in different data registers, the code generator generates the following M68000 code:

```
movl    d6,d0    | int2 → d0
addl    d5,d0    | int2 + int3 → d0
movl    d0,d7    | int2 + int3 → int1
```

c2 finds a register-copy instruction, `movl d0,d7`, and if the source register (*d0* in this case) is dead (i.e. has no use before being redefined) after this instruction, it scans backward within the basic block to find the last instruction which defines but does not use the source register (in this case, `movl d6,d0`). Then, if the target register *d7* is not used or defined in between these two instructions, it turns them into

```
movl    d6,d7
addl    d5,d7
```

Similar code is generated and then optimized by *c2* for three-address FPA instructions. *c2* actually looks for many different patterns of register usage and turns them into more efficient ones. Patterns can be added to *c2* relatively easy.

6. Code Quality and Optimizer Performance

Our experience shows that the global and peephole optimizers together make FORTRAN programs run from 10% to 80% faster than the unoptimized code. Code size is reduced by about 10% on the average.

An example of code quality of which we are especially proud is the routine in the Linpack benchmark [Dong86] which accounts for the vast majority of its execution time. The following loop is one drawn from that routine. While not the one which takes the largest fraction of the time, it shows more interesting optimizations.

```
do 10 i = 1,n
  dy(iy) = dy(iy) + da*dx(ix)
  ix = ix + incx
  iy = iy + incy
10 continue
```

The code our compiler generates for the double-precision version for a Sun-3 with an FPA and with optimization disabled is

```

L33:
    movl    a6@(0x10),a0
    movl    a6@(0xc),a1
    fpmoved a1@,fpa1
    fpmuld  a0@(-0x8,d5:1:8),fpa1
    movl    a6@(0x18),a0
    fpadddd a0@(-0x8,d6:1:8),fpa1
    movl    a6@(0x18),a0
    fpmoved fpa1,a0@(-0x8,d6:1:8)
    movl    a6@(0x14),a0
    movl    a0@,d0
    addl    d0,d5
    movl    a6@(0x1c),a0
    movl    a0@,d0
    addl    d0,d6
    addql   #0x1,d7
    subql   #0x1,a6@(-0x8)
    jpl     L33

```

With the optimizer enabled the resulting code is

```

L77016:  fpmul3d@3  a4@,fpa4,fpa0
        fpadddd@3 a5@,fpa0
        addl     d6,a4
        fpmoved@3 fpa0,a5@
        addl     d5,a5
        dbra     d7,L77016
        clrw     d7
        subql    #1,d7
        jpl     L77016

```

which is the best possible code for the loop. Our compiler also generates optimal code for the simpler loop which accounts for the largest fraction of the execution time.

Table 1 shows execution times in seconds for several integer benchmarks compiled with the Release 3.0 optimizing FORTRAN compiler. The Release 2.0 compiler is the 4.2 BSD FORTRAN with several improvements. The Sun-2/50 is based on a 10 MHz Motorola MC68010 CPU and the Sun-3/75 on a 16.67 MHz MC68020.

The benchmarks are as follows:

- | | |
|------------|---|
| Sort | A bubble sort of an array of 1000 integers. |
| S1 and NS1 | Two subroutines which measure subscripting and looping and which are discussed at length in [Much86]. |
| Sieve | The sieve of Eratosthenes, which computes the primes less than 8192, iterated 100 times. |

Table 2 displays floating-point performance on the Linpack and Whetstone benchmarks with the Floating-Point Accelerator and the Release 3.1 FORTRAN compiler (the first release to support the FPA).

Processor/ Compiler	Optimized?	Sort	S1	NS1	Sieve
Sun-2/50 Release 2.0	No	20.9	199.9	199.9	44.5
Sun-2/50 Release 2.0	Yes	11.6	66.8	99.9	30.5
Sun-2/50 Release 3.0	Yes	4.9	56.5	61.5	20.8
Sun-3/75 Release 3.0	Yes	1.3	12.8	13.8	5.0
VAX-11/780 VMS FORTRAN	Yes	—	45.1	115.0	—
VAX-11/785 UNIX FORTRAN	Yes	—	48.0	75.0	—

Table 1. Integer Performance (in seconds)

The VAX-11/780 Linpack numbers are for a version with the inner loops hand-coded in assembly language. On some machines, including the VAX, this makes a significant difference in performance. As noted above, it does not for the Sun-3.

Processor/ Compiler	Optimized?	Linpack		Whetstone	
		Single (Kflops)	Double (Kflops)	Single (KWhets)	Double (KWhets)
Sun-3/75 FPA	No	258	184	1900	1280
Sun-3/75 FPA	Yes	615	405	2600	1800
MicroVAX II VMS FORTRAN	Yes	98	43	925	680
VAX-11/780 FPA, VMS	Yes	340	170	1340	780

Table 2. Floating-Point Performance

7. Aliasing

The *aliaser* deals with the problem of aliases arising from the presence of multiple names mapping to the same memory areas. The mechanism which maps variables to storage locations during the execution of a program has a strong effect on creation of aliases. This mapping depends on the language in which the program is written and results in different forms of aliasing.

In FORTRAN, the set of names which may refer to the same location may be determined exactly at compile time. This is known as *static aliasing* and is handled by noting which leaves can potentially overlap. An overlap list is computed for each leaf by the compiler front end and passed to the *aliaser*.

Languages such as C and Pascal pose an additional problem for the optimizer beyond that posed by FORTRAN, namely *dynamic aliasing*. In C, aliases may arise from memory overlaps, array references, use of pointer variables, expressions that compute pointers, and procedure and function calls. Aliases in Pascal programs arise for similar reasons as in C, but are somewhat easier to handle since pointers may only point to objects of the same types and only into the heap.

Dynamic aliases are those that cannot be determined exactly at compile time. The necessity for computing dynamic aliases is demonstrated by the C routine shown in Figure 3. The second computation of *k* seems to be redundant. However, if *q* had been set to point to *a* by a previous statement, then **q = 13* redefines *a* and thus the second computation of *k* is not redundant. Note that whether *q* actually will point to *a* at runtime cannot, in general, be determined — only whether it might point to it.

```
example( )
{   int a, k, *q;

    . . .

    k = a+5;
    *q = 13;
    k = a+5;

    . . .
}
```

Figure 3. An Example of Dynamic Aliasing in C

In Sun IR storage references which go indirect through an address expression are limited to occurring in IFETCH, ISTORE, INDIRREF and PARAM triples. Dynamic aliasing is handled by the *aliaser* module by ensuring that each such triple has a valid can-access list which identifies the objects which might be accessible through it. It uses a list of sources (i.e. pointers) and a list of targets (i.e. externals and locals whose addresses have been taken) computed by the compiler front end to perform its analysis.

The *aliaser* begins by scanning the triples making up a procedure, to determine for each pointer used what it could point to at that location in the code. Then, for each basic block, it computes by iteration what each pointer could point to within the block. Using this information, along with the overlap lists for the leaves, it determines a valid can-access list for each triple.

The information in the can-access list is made explicit for later use by the optimizer. For example, if indirection through *p* can access *a* and *b*, then an IFETCH through *p* is accompanied by IMPLICITUSE triples for *a* and *b*.

Due to the absence of interprocedural data flow analysis [Bann78,Bart78], calls are dealt with as a worst case, i.e. they are accompanied by `IMPLICITUSE` and `IMPLICITDEF` for all externals and locals whose addresses have been taken.

8. Conclusion and Future Plans

This section describes problems encountered in developing *iropt* and how they were solved, along with plans for its future development.

For the most part implementing *iropt* was straightforward. Most problems were associated with achieving acceptable compilation speed. A pervasive problem was and, in some instances, continues to be heavy paging caused by large heap storage requirements. The solution was to redesign several key data structures to make them significantly smaller and to write a specialized heap manager to provide better locality of memory references. The approach used was to partition the heap into large chunks, called allocation blocks and to maintain a collection of queues, one for each frequently allocated structure type. Each queue allocates structures from its current allocation block and acquires a new block when necessary. At appropriate points in the processing the queues are emptied and their allocation blocks are returned to the free pool.

Another problem area is making side effects of calls explicit. Currently, we assume that all externals and locals whose addresses are taken may be defined or used by a call. This both uses a lot of storage and results in unduly pessimistic interprocedural optimization. The excessive storage usage is handled by providing two flags to control optimization: `-O` provides full optimization but, in some cases, requires inordinately long compilation times because of excessive paging, while `-P` provides partial global optimization in return for much less time spent in optimization. Doing better optimization across calls is an area for future enhancement.

Areas planned for future enhancement include loop unrolling, interprocedural flow analysis, improvements to strength reduction of multiple subscripts, and the incremental updating of data flow information [Ryde83,Ghod83].

Acknowledgements

We thank Alastair Fyfe and Christopher Aoki for their contributions to the design and development of the global optimizer.

References

- [AdaM85] Adams, E. & S.S. Muchnick, Dbxtool—A Window- and Mouse-Based Symbolic Debugger, *Proc. of the 1985 Summer Usenix Conf.*, Portland, OR, June 1985, pp. 213 - 228.
- [AhoS86] Aho, A.V., Sethi, R. & J.D. Ullman, **Compilers: Principles, Tools, and Techniques**, Addison-Wesley, Reading, MA, 1986.
- [Bann78] Banning, J.P., A Method for Determining the Side Effects of Procedure Calls, Ph.D. dissertation, Dept. of Computer Science, Stanford University, 1978.

- [Bart78] Barth, J.M., A Practical Interprocedural Data Flow Analysis Algorithm, *Comm. of the ACM*, vol. 21, no. 9, Sept. 1978, pp. 724 - 736.
- [CocK77] Cocke, J. & K. Kennedy, An Algorithm for Reduction of Operator Strength, *Comm. of the ACM*, vol. 20, no. 11, Nov. 1977, pp. 850 - 856.
- [CouH86] Coutant, D.S., C.L. Hammond & J.W. Kelly, Compilers for the New Generation of Hewlett-Packard Computers, *Hewlett-Packard Journal*, vol. 37, no. 1, Jan. 1986.
- [Dong86] Dongarra, J.J., Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment, Tech. Memo. 23, Argonne National Lab., Argonne, IL, 17 Feb. 1986.
- [Ghod83] Ghodssi, V., Incremental Analysis of Programs, Ph.D. dissertation, University of Central Florida, 1983.
- [GibM86] Gibbons, P.B. & S.S. Muchnick, Efficient Instruction Scheduling for a pipelined Architecture, *Proc. of SIGPLAN Symp. on Compiler Constr.*, Palo Alto, CA, June 1986.
- [Lion79] Lions, J., *The Second Pass of the Portable C Compiler*, AT&T Bell Laboratories, June 1979.
- [Much86] Muchnick, Steven S., Here Are (Some of) the Optimizing Compilers, *SIGPLAN Notices*, vol. 21, no. 3, March 1986, pp. 11 - 15.
- [MucJ81] Muchnick, Steven S. & Neil D. Jones [eds.], **Program Flow Analysis: Theory and Applications**, Prentice-Hall, 1981.
- [Ryde83] Ryder, B.G., Incremental Data Flow Analysis, *Proc. of Tenth ACM Symp. on Princ. of Prog. Langs.*, Austin, TX, Jan. 1983, pp. 167 - 176.
- [Spil71] Spillman, Thomas C., Exposing Side-Effects in a PL/I Optimizing Compiler, *Information Processing 71*, North-Holland, 1972, pp. 376 - 381.
- [Weih80] Weihl, William E., Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables, and Label Variables, *Proc. of Seventh ACM Symp. on Princ. of Prog. Langs.*, Las Vegas, NV, January 1980, pp. 83 - 94.

Appendix A. An Example of Sun IR

The following is an example of a C routine (Figure 4) and the Sun IR generated for it (Figure 5).

```
main( )
{
    int a, b, *p, *q;

    b = 2;
    a = b + 5;
    if (b > 0) {
        p = &b;
        q = &a;
    }
    a = *p;
    *q = b;
}
```

Figure 4. An Example C Program

```
SEGMENTS
ARG_SEG      length 0   STG_SEG  LCLSTG_SEG
VAR_SEG12    length 0   STG_SEG  LCLSTG_SEG
ARR_SEG12    length 0   STG_SEG  LCLSTG_SEG
DATA_SEG     length 0   STG_SEG  LCLSTG_SEG
AUTO_SEG     length 0   STG_SEG  LCLSTG_SEG
HEAP_SEG     length 0   STG_SEG  EXTSTG_SEG
DREG_SEG     length 0   REG_SEG  LCLSTG_SEG
AREG_SEG     length 0   REG_SEG  LCLSTG_SEG
(null)       length 0   REG_SEG  LCLSTG_SEG

BLOCKS
[0]          main label 14 next 1
triples:    0 1 2 3 4 5 8 9 10 11 12 13 14 15 16 17 19
[1]         label 13 next -1
triples:    20 22
```

Figure 5. Sun IR Generated for the C Routine in Figure 4
(before optimization)

LEAVES

[0]	q	size 4	*to int	VAR AUTO_SEG(14,-16)
[1]	p	size 4	*to int	VAR AUTO_SEG(14,-12)
[2]	""	size 4	int	CONST 17
[3]	""	size 4	bool	CONST true
[4]	""	size 4	bool	CONST false
[5]	""	size 4	int	CONST 16
[6]	""	size 4	int	CONST 5
[7]	a	size 4	int	VAR AUTO_SEG(14,-4)
[8]	""	size 4	int	CONST 2
[9]	b	size 4	int	VAR AUTO_SEG(14,-8)
[10]	""	size 11	string	CONST #PROC# 04
[11]	""	size 4	int	CONST 14
[12]	""	size 4	int	CONST -1
[13]	""	size 4	int	CONST 0
[14]	""	size 4	int	CONST 13

TRIPLES

[0]	labeldef	L[11]
[1]	pass	#PROC# 04
[2]	assign	b L[9] L[8]
[3]	int plus	b L[9] L[6]
[4]	assign	a L[7] T[3]
[5]	int gt	b L[9]
[6]	labelref	L[2] L[3]
[7]	labelref	L[5] L[4]
[8]	cbranch	T[5] labels: L[2] if L[3] L[5] if L[4]
[9]	labeldef	L[2]
[10]	string addrof	b L[9]
[11]	assign	p L[1] T[10]
[12]	string addrof	a L[7]
[13]	assign	q L[0] T[12]
[14]	labeldef	L[5]
[15]	int ifetch	p L[1]
[16]	assign	a L[7] T[15]
[17]	istore	q L[0] b L[9]
[18]	labelref	L[14] L[13]
[19]	goto	L[14]
[20]	labeldef	L[14]
[21]	labelref	L[12] L[13]
[22]	goto	L[12]

Figure 5 (cont.). Sun IR Generated for the C Routine in Figure 4
(before optimization)


```

SEGMENTS
ARG_SEG      length 0   STG_SEG   LCLSTG_SEG
VAR_SEG12    length 0   STG_SEG   LCLSTG_SEG
ARR_SEG12    length 0   STG_SEG   LCLSTG_SEG
DATA_SEG     length 0   STG_SEG   LCLSTG_SEG
AUTO_SEG     length 0   STG_SEG   LCLSTG_SEG
HEAP_SEG     length 0   STG_SEG   EXTSTG_SEG
DREG_SEG     length 0   REG_SEG   LCLSTG_SEG
AREG_SEG     length 0   REG_SEG   LCLSTG_SEG
(null)       length 0   REG_SEG   LCLSTG_SEG

LEAVES
[0]    q      size 4   *to int   VAR AUTO_SEG(14,-16)  references: 5 8
[1]    p      size 4   *to int   VAR AUTO_SEG(14,-12)  references: 2 7
[2]    ""     size 4   int       CONST 17
[3]    ""     size 4   bool      CONST true
[4]    ""     size 4   bool      CONST false
[5]    ""     size 4   int       CONST 16
[6]    ""     size 4   int       CONST 5
[7]    a      size 4   int       VAR AUTO_SEG(14,-4)   references: 3 4
[8]    ""     size 4   int       CONST 2
[9]    b      size 4   int       VAR AUTO_SEG(14,-8)   references: 0 1
[10]   ""     size 11  string    CONST |#PROC# 04
[11]   ""     size 4   int       CONST 14
[12]   ""     size 4   int       CONST -1
[13]   ""     size 4   int       CONST 0
[14]   ""     size 4   int       CONST 13
      POINT 15

BLOCK [0]   main label 77004   next 1 loop_weight 1 pred: succ: 2 3
                                dfonext 3
                                [0]   labeldef      B[0]
                                [1]   pass           |#PROC# 04
                                [1]   assign         b L[9] L[8] canreach: T[8] var refs: 0
                                [2]   int gt         L[8] L[13]
                                [4]   cbranch        T[2] labels: B[3] if L[3] B[2] if L[4]

BLOCK [1]   label 77001       next 2 loop_weight 1 pred: 2 succ:
                                dfoprev 2
                                [5]   labeldef      B[1]
                                [6]   goto          exit

```

Figure 6. Sun IR Generated for the C Routine in Figure 4
(after optimization)

```

BLOCK [2]  label 77002  next 3 loop_weight 1 pred: 0 3 succ: 1
                        dfonext 1 dfoprev 3
[7]  labeldef          B[2]
[8]  implicit_use      b L[9]T[9] reachdef1: T[1]  var refs: 1
[9]  int ifetch        p L[1] reachdef1: T[17]  var refs: 2
[10] assign            a L[7]T[9] canreach: T[11]  var refs: 3
[11] implicit_use      a L[7]T[12] reachdef1: T[10]  var refs: 4
[12] istore            q L[0] L[8] reachdef1: T[19]  var refs: 5
[13] implicit_def      a L[7]T[12] var refs: 6
[14] goto
[18] labelref          B[1] L[13]

BLOCK [3]  label 77003  next -1 loop_weight 1 pred: 0 succ: 2
                        dfonext 2 dfoprev 0
[15] labeldef          B[3]
[16] int addrof        b L[9]
[17] assign            p L[1]T[16] canreach: T[9]  var refs: 7
[18] int addrof        a L[7]
[19] assign            q L[0]T[18] canreach: T[12]  var refs: 8
[20] goto
[23] labelref          B[2] L[13]

```

Figure 6 (cont.). Sun IR Generated for the C Routine in Figure 4
(after optimization)

Four Generations of Portable C Compiler

David M. Kristol

AT&T Information Systems
Summit, NJ 07901

ABSTRACT

In nearly ten years' time the original Portable C Compiler (**PCC**) has evolved through three further generations that have improved its speed, generality, and code quality. Each generation uses a different code generation strategy. These compiler technologies, named **PCC2**, **QCC**, and **RCC** are all available in current AT&T C compilation system products, notably those for the AT&T 3B2 computer, which is based on the WETM 32100 microprocessor.

The C language recognizer, or *front end*, part of **PCC2**, **QCC**, and **RCC** is identical. (The code is common among them.) It is essentially the same front end as that used in **PCC** except that it has better error checking and greater robustness. The principal differences, then, are in the respective code generation strategies, and they are the topic of this paper.

INTRODUCTION

The Portable C Compiler has been the base for uncounted numbers of C compilers on a wide variety of machines in its nearly ten years' existence. The reasons for its success are simple. Steve Johnson, author of both **PCC** and **PCC2**, put it this way: "The compiler is efficient enough, and produces good enough code, to serve as a production compiler."^[1] (The C compilers that were part of UNIX* System V, Release 2, were **PCC**-based.) What's more, **PCC**'s portability means that "if you need a C compiler written for a machine with a reasonable architecture, the compiler is already three quarters finished."^[1]

Compiler development within AT&T did not cease with **PCC**. We have continued to investigate and to use new compiler technology. After reviewing some of the internal details of **PCC**, this paper will describe some of the newly developed technology that has found its way into compilers for, among other machines, the WETM 32100 microprocessor, the heart of the AT&T 3B2 computer.

The improvements that I will describe concern the strategies that the compiler uses to generate code. For the rest of this paper I will assume that the parser portion of the compiler has produced a tree-structured representation of C expressions, and that the code generator must produce the assembly language that will realize the expression's semantic intention. The new technologies, **PCC2**, **QCC**, and **RCC**, share a C language parser that is closely related to **PCC**'s, except for improved correctness and error detection.

THE PROBLEM

Code generation is easy when a machine has an infinite number of general registers, each of which may hold any datum. Of course, there are no real machines that meet this ideal, or that even come close. (That's one reason there is still a market for compiler writers.) Architecture designs have a finite, usually small, number of machine registers, and sometimes certain functions are tied to specific registers. Because machine registers must be considered scarce resources, how they are allocated, and in what order they are used, has a significant effect on

* UNIX is a trademark of AT&T.

the quality of code that the compiler generates. The PCC Family history is a collection of solutions to the register allocation and code ordering problem.

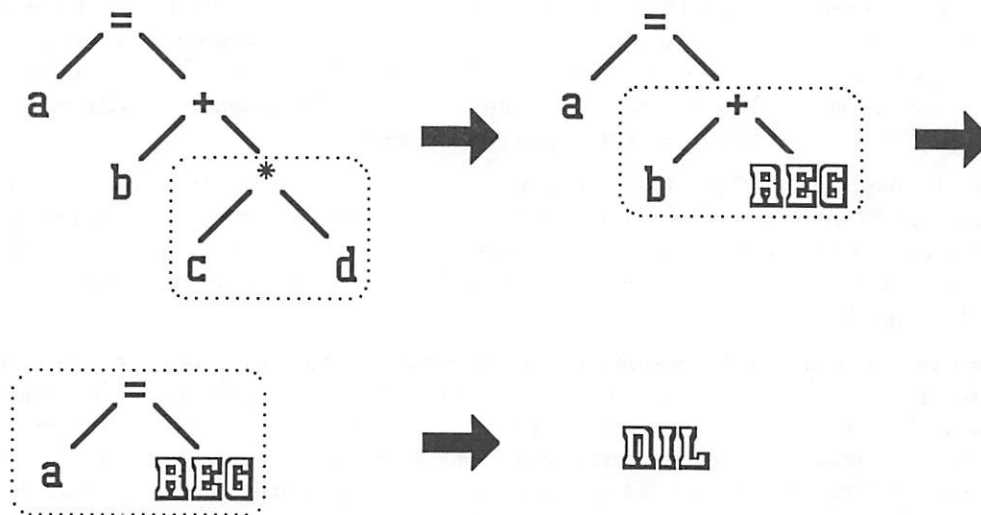
PCC FAMILY ARCHITECTURE

The PCC Family compilers have two main pieces: a (relatively) machine-independent front-end, or parser, that builds what are called *C-trees*; and a table-driven interpretive back-end, or code generator that is machine-dependent. To create a new compiler instance, an implementor provides definitions for symbols that describe the sizes and alignments of various data types, the byte ordering and the stack layout of the target machine, and a set of *templates* that describes what code the compiler must generate for various C operators. The templates contain the C operator and *shapes* that describe each of the operator's operands. Shapes are most easily thought of as the C-trees that correspond to an architecture's address modes.

The PCC Family back-ends consume the C-trees by starting at the fringe of the tree and working back to the root. Example 1 shows how this process works on the simple expression

`a = b + c * d;`

(REG represents a register node in the tree.) Part of the tree's fringe is matched by a template, the corresponding assembly code is emitted, and the portion of the tree that was matched is rewritten to be the template's result, if any, usually a scratch register. Thus the code generation and tree rewriting mirrors the execution-time behavior of the code that is generated.



Example 1. Reduction of a C expression.

PCC — THE PORTABLE C COMPILER

Steve Johnson wrote the Portable C Compiler, PCC, in 1977 as part of an experiment to port UNIX from a PDP-11[†] base to an Interdata 8/32 computer.^[2] It became possibly the most widely

used compiler that can generate native machine code. **PCC** established the compiler design framework that continues in the **PCC** Family.

PCC Code Generation

PCC's code generation is based on theoretical work by Sethi and Ullman^[3] and others. This work describes how code generation may be ordered to use the smallest possible number of machine registers. In ^[1], Johnson describes how **PCC** partitions the code generation problem into determining the proper order and then following it.

The compiler implementor supplies a function to calculate the so-called Sethi-Ullman (or SU) numbers. The machine-independent part of the back-end calls this routine to populate an expression tree with SU numbers that indicate how many registers would be needed to do that part of the tree. The number at the root of the tree must be no greater than the number of scratch machine registers. Otherwise there are too few registers, and the compiler must rewrite (*spill*) part of the tree so its result will be in a temporary location, rather than in a machine register. Once the compiler successfully populates the tree with SU numbers such that no number is greater than the number of scratch registers, code generation proceeds. The code generator walks down the tree, doing higher SU numbers before lower, until it reaches leaves, at which point it begins emitting code.

This code generation approach works well most of the time. However, the Sethi-Ullman computation routine tends to look like black magic, since when it assigns numbers of registers for computations it must anticipate what templates will be used at any point and what tree rewrites will occur. When templates get added, removed, or just moved, the SU computation often has to be modified to correspond.

Rarely could the SU computation be completely accurate, so it had to strike a careful balance between optimism and pessimism. On one hand, if the SU computation is too pessimistic, the generated code will be low quality, with many spills. On the other hand, if it is too optimistic, the back-end can discover that fewer registers are available than the SU number promised, and it will quit with a compiler error.

Another weak spot in **PCC**'s design was the handling of shapes. When **PCC** was designed, "indirect through the sum of a register and offset" was considered an exotic address mode. This address mode, which came to be known as an *OREG*, is a good description for such things as stack locations and structure references, where the pointer to the structure is in a register. Each time it emits code, **PCC** must reexamine the tree to recognize if an *OREG* has resulted from the rewrite. When fancier architectures emerged, like the VAX and Motorola 68000[†], that had double indexing, **PCC** compilers had to scramble to fit such indexing into the *OREG* mold.

To summarize, **PCC** was the first practical, portable compiler technology that became widely available. It can generate good quality code, although making it do so is challenging for the implementor.

PCC2 — CHILD OF PCC

Aho and Johnson did further work on code generation theory^[4], and in 1978 Johnson produced a revised **PCC**, which became known as **PCC2**. The major changes from **PCC** were these:

- Code generation tree matching occurs top-down.

† PDP and VAX are trademarks of Digital Equipment Corporation.

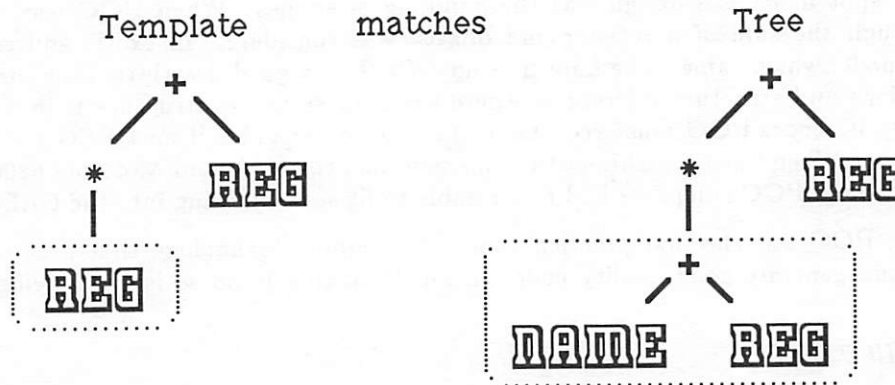
1. The AT&T UNIX PC is based on a Motorola 68010.

- Implementor-specified costs control code generation.
- Templates may contain arbitrary, implementor-specified shapes.
- The implementor uses a descriptive file to specify templates, shapes, and costs.

The Aho and Johnson theory states that the cost of a computation may be partitioned into the cost of an operation and the cost of obtaining the operands. This may apply recursively, because obtaining the operands may entail further computation. To minimize the cost of a computation, one need only minimize the sum of the cost of the operands and the operation at each step.

The **PCC2** back-end walks the expression tree, and, using the implementor supplied costs, it identifies the least expensive way to generate code for it, using a dynamic programming algorithm. Since the cost calculation algorithm considers the templates that would actually be used at each step, the calculated costs are accurate. Furthermore, the cost calculations take into account the number of registers available at each step, so register allocation is a by-product. Register spilling is another by-product because its cost is considered as a possibility at each step. This approach solves the problem of the separation between SU computation and code generation from which **PCC** suffered. However, the dynamic programming theory only allowed for one type of register, a shortcoming that was later addressed by **RCC**.

PCC2 introduces the idea of treating register shapes as *wildcards*. During its top-down tree match, **PCC2** attempts to match a template against the expression tree. If it finds that a register in a shape of the template matches a non-register in the tree, **PCC2** considers the result to be a (wildcard) match, with the following proviso: it must now calculate the *sub-tree* whose root is the mismatched non-register and assume that the result of that computation will end up in a register. At that point the register node in the shape will indeed match a register, and code generation for the original template may proceed. Example 2 depicts the case of a register (REG) node in a template matching part of a tree as a wildcard. The NAME node represents an external variable.



Example 2. Register matches as wildcard.

The theory behind **PCC2** assumes that all registers are equivalent and can contain any datum. Indeed, using registers in shapes as wildcards reinforces that need. However, by carefully designing templates and the supporting machine-dependent code, implementors have been able

to create **PCC2** compilers for machines that do not conform to this restriction, such as the Motorola 68000 and Intel 8086².

The machine description file that Johnson introduced for **PCC2**, commonly called *stin* (shape and template input), is much more concise and easy to maintain than the previous **PCC** template representation, an initialization of a large array of structures that had to be maintained by hand. *Stin* descriptions are processed by a program to produce the **PCC2** equivalent of the **PCC** array of structures.

PCC2 compilers produce locally excellent code, but they do so slowly because of the cost calculations, which take two passes through each expression tree. In the first pass the back-end determines the least-cost code sequence. In the second pass, the back-end follows that sequence and generates code accordingly.

PCC2's *stin* file led to better compiler maintainability than **PCC** afforded. Within my group at AT&T we wanted to replace all of our **PCC**-based compilers with **PCC2**-based ones. However, because **PCC2** was as much as three times slower, we could not justify such an impact on our customers. We therefore sought to improve **PCC2**'s speed and still retain its other advantages. The result was **QCC**.

QCC — A FASTER PCC2

QCC is based on a principle that arose from a simple observation: most of the time **PCC2** explores a great many possible code generation sequences before it chooses the obvious one. In other words, most of the time the cost-based algorithm is unnecessary. Rob Murray and I therefore replaced the cost-based algorithm with one that made the obvious choice, when there was one. The only time that a choice was "un-obvious" was when the algorithm needed more scratch registers than it had available, in which case it chose a simple spill-to-temporary algorithm and tried again.

The algorithm is a top-down tree match like **PCC2**'s. Unlike **PCC2** however, **QCC** uses the first template (*i.e.*, operation) for which the operand shapes (*i.e.*, address modes) match. We also chose the first matching shape. A register node in a shape, if it doesn't actually match a register in the C-tree, is treated as a *wildcard*, as in **PCC2**. **QCC** knows that before it can use a template that has wildcard registers in a shape, it must generate the instructions that will actually load those registers. At that point it descends the tree to generate that code first.

Two important observations must be made here. First, since we choose the *first* matching template, the order of those templates is important. In **PCC2**, you will recall, the compiler chose the lowest cost alternative from all alternatives, so the order of templates was irrelevant.

The second observation regards shapes. It would seem beneficial for **QCC** to match as large a piece of a tree as possible with a shape. Implicit in that assumption is that a machine's hardware can do the operations (notably address addition) better (and cheaper) than a sequence of equivalent instructions. This matching of larger shapes in preference to smaller ones has been called "maximal munch," because the compiler tries to bite off as large a piece of the expression tree as possible.

To get the most from **QCC**'s first-match algorithm, then, the compiler must examine templates and shapes in a preferred order. Because the description of templates and shapes was already embodied in the *stin* file, and that file was already processed by a program (called *sty*), it was natural to enhance *sty* to order the shapes and templates. In fact, one original design goal for **QCC** was to be able to take a **PCC2** compiler and convert it to a **QCC** compiler with a few

2. The Intel 8086 is the processor in the AT&T PC 6300.

quick strokes.

That goal was not entirely met, and the reasons were both good and bad. On the good side, we could do some things in **QCC** that were impossible in **PCC2**, and that reduced the number of templates an implementor had to write and simplified others. On the bad side, *sty* was not 100% successful at ordering templates, and a little hand tuning often improved code quality.

On balance, the **QCC** experiment was a tremendous success. **QCC** compilers actually turned out to be faster than equivalent **PCC** compilers. The quality of generated code fell between that of **PCC** and **PCC2**, but very close to the higher quality **PCC2** side. It still suffered from one shortcoming of **PCC2**: it could handle only one kind of machine register.

RCC — QCC WITH REGISTER SETS

The restriction of **QCC** to one kind of machine register threatened to hamper its usefulness. As **QCC** was emerging, so was the WE 32106 Math Acceleration Unit (MAU) floating point co-processor for the WE 32100 microprocessor. Like many such co-processors, the MAU has its own set of registers, and they are emphatically *not* general purpose. They are for floating point calculations. Although we felt that we could use various tricks in **QCC** to generate code for the 32100/32106 combination, we preferred a more direct approach. This motivation led to my development of **RCC**.

RCC is **QCC** with more register bookkeeping. Where **QCC** keeps track of *how many* registers it has available to it as it descends a C-tree to generate code, **RCC** keeps track of *which ones*, too. In other words, **RCC** does a heuristic register allocation “on the fly,” during tree matching.

An **RCC** implementor must divide a machine’s registers into two groups (as with **PCC**, **PCC2**, and **QCC**): *scratch* and *user*. *User* registers are the ones that are available to the C programmer as *register* variables. The compiler manages and allocates the scratch registers for intermediate computations. Beyond the user/scratch division, an implementor is free to partition the registers in any useful manner. For example, on the 32100/32106 combination, three CPU and one MAU register are designated as scratch, and six CPU and two MAU registers are designated as user registers.

RCC also extends the notion of *wildcard* a bit further. In **RCC**, a register node in a tree might match a register node in a shape as a *wildcard* if the register in the tree is not one of the ones expected by the shape. For example, an implementor for an Intel 80286³ compiler can write a template in which the shape corresponding to the shift count is precisely register **CX**. **RCC** guarantees that if that template gets used, **CX** will, in fact, contain the shift count. If the result of a shift count computation ends up in **AX**, **RCC** will see to it that the value gets moved to **CX**. Moreover, **RCC** will actually attempt to have the computation leave its result in **CX** in the first place.

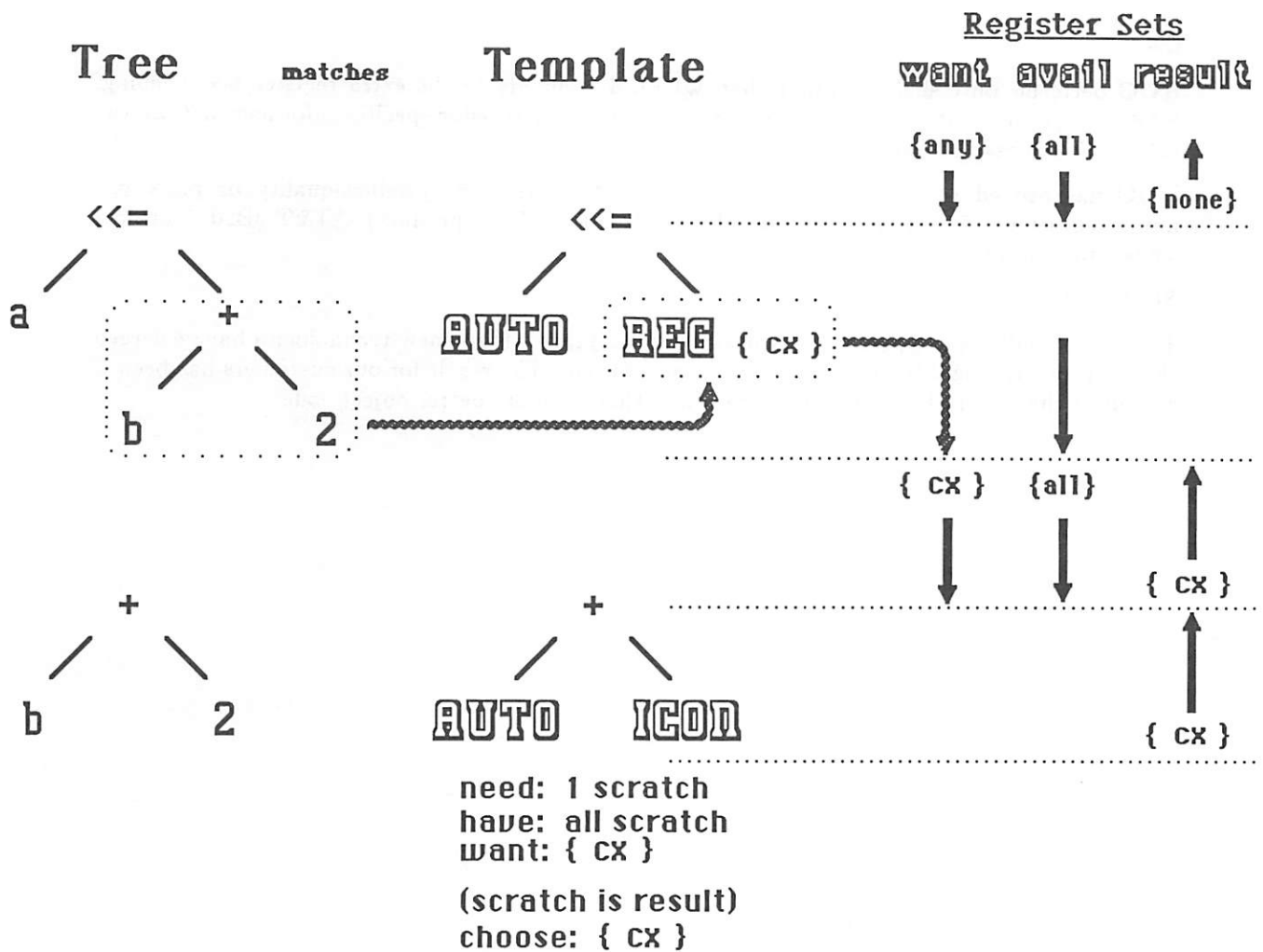
Example 3 shows how **RCC** generates code for the expression

```
a <<= b + 2;
```

for the Intel 80286, where **a** and **b** are automatic variables. As I stated above, the shift count must be in register **CX**, and the first template shows that the right operand indeed must be **CX**.

The rest of the example shows the register accounting that takes place. *Want* and *available* sets of registers are passed down the tree during code generation, and a *result* set of registers gets

3. The AT&T PC 6300+ uses the Intel 80286 processor.



Example 3. RCC Register Accounting

passed upward. Thus, when the back-end goes to generate code for the `+` operations, it knows that `CX` is the preferred result register. Since the template must allocate one register for its computation; since that register becomes the result of the addition; and since `CX` is available, **RCC** chooses `CX` as the scratch (and result) register for the `+`. At the next higher level, the pre-condition for the `<<=` template, loading `CX`, has been met, and code for the shift may be generated.

RCC lets the compiler implementor specify numbers and types of scratch registers with the same precision as registers in shapes may be specified. For example suppose a scratch register is needed for some calculation on the 32100/32106 combination. Should it be a general purpose (32100) register or a MAU (32106) floating point register? **RCC** can't guess the right answer, but the implementor can say explicitly which one to use with an appropriate notation in the *stin*

file.

RCC performs only slightly slower than QCC, due mainly to the extra register bookkeeping. RCC's version of *sty* knows how to process the extra register-specific information that the RCC *stin* file may contain.

RCC has proved to be quite versatile. It has been used for product-quality or prototype implementations of compilers for the AT&T 3B2/400 (C-FP+ product), AT&T 3B20, Motorola 68000, and Intel 80286.

SUMMARY

The PCC Family of compilers has evolved over ten years. Three new technologies have emerged that improve the quality and speed of code generation. The result for our customers has been C compilers that are faster and more reliable and that generate better object code.

REFERENCES

1. *A Tour Through the Portable C Compiler*, in **Documents for UNIX**, Volume 2. Bell Laboratories. January, 1981.
2. Johnson, S.C., and Ritchie, D.M. "Portability of C Programs and the UNIX System." *The Bell System Technical Journal*, Vol. 57, No. 6, Part 2, July-August, 1978.
3. Sethi, R. and Ullman, J.D. "The Generation of Optimal Code for Arithmetic Expressions." *J. ACM* **17**,4, October, 1970.
4. Aho, A.V. and Johnson, S.C. "Optimal Code Generation for Expression Trees," in *Proceedings of the ACM Symposium on the Theory of Computing*. 1975.

The Notifier

Steve Evans

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
(415) 960-1300
sun!sevens

ABSTRACT

The *notifier* is an event-driven dispatcher. It is a general-purpose library package for distributing events to a collection of clients within a process. The notifier detects events that its clients have expressed an interest in and dispatches these events to the proper clients, queuing client processing so that clients respond to relevant events in a predictable order.

The paper motivates the design of the notifier and describes the flavor of the programming interface to it.

1. BACKGROUND

Traditional UNIX programs make the management of the main control loop the application's responsibility. In the quest for an easy to use, yet powerful, programming interface to SunView (a multi-window user interface tool kit),¹ this traditional approach breaks down. A typical SunView application is made up of many subsystems:

- Each application has a frame manager that provides a window name stripe, icon and border. The frame manager needs to know when a signal (SIGWINCH) arrives so that it knows when to repaint itself. The frame manager needs to know when there is input pending for it on a file descriptor so that it can interact with the user to provide window management feedback. There may be multiple frames active on the screen that are controlled by a single user process.
- Each frame is subdivided into one or more regions that are usually managed by separate library packages. These regions managers are concerned with getting user input and repainting themselves just like frames. Examples of regions include control panels, text managers, terminal emulators, generic drawing canvases and scrollbars. Some regions provide a blinking caret or some other form of timeout dependent feedback and thus need to be concerned with getting control at periodic intervals. Still other regions, e.g., the terminal emulator region, read from and write to pipes. They are concerned with finding out when input is available on a pipe and when a pipe has been drained of output.
- There is a subsystem that supports what one could call the "cut and paste" paradigm. It is concerned with remote procedure call communications with other processes. Thus, it must worry about a variety of socket management activities and be able to make connections with other processes.
- All of the subsystems described above are implemented as library packages. Application supplied code may be concerned with similar as well as additional control concerns.

The purpose of the above list is to emphasize the notion that asking application level code to support this structure of subsystems would be disastrous. Asking an application programmer to know when and what to dispatch to these subsystems would be a nightmare. Unfortunately, UNIX and the C library were not designed to support this level of complexity and diversity within a single user process.

There are at least two approaches to this problem. One approach is to provide a lightweight process mechanism² that gives every subsystem the impression that it owns the thread of control in the user process

and at the same time allows multiple threads in the same address space. A lightweight process environment provides mechanisms for managing parallelism and concurrency. Lightweight processing is particularly good at multiplexing CPU usage among computationally intensive tasks.

A second approach to the problem of flow of control management is to provide a centralized “interrupt” manager, and is the topic of this paper. The *notifier* allows its clients to share the same address space and a single thread of control among multiple clients, thus multiplexing the tasks within a user process. Control is routed through the notifier and distributed to its clients via procedure call-outs. The notifier is particularly good at multiplexing CPU usage among IO intensive tasks.

The notifier’s control mechanism is hidden from direct view of the application and thus provides a degree of implementation and information hiding for each of the clients of the notifier. The result is simpler application code then would be obtained by having application level code detect and dispatch all events to all clients.

The programming style utilized by notifier clients is *notification-based*, meaning that notifier clients preserve their state in a client specific state machine and are driven from state to state via procedure calls from the notifier.

With the notifier, existing non-notification-based programs that maintain their state in the program counter and on the stack need not invert their control structure in order to run in the notifier environment. This is important because converting an existing program’s flow of control to be notification-based is often very difficult. The notifier would not be a viable approach if it discouraged programmers from moving their applications into the notifier environment. For example, one would not want to rewrite Emacs if one were to construct an application with a frame, control panel and Emacs region.

1.0.1. When to use the notifier

A program that uses SunView is already implicitly using the notifier; the notifier is used by SunView library subsystems. Notifier features that the library subsystems utilize are available to application level code as well. These features include the ability to:

- Catch signals, e.g., SIGCONT.
- Notice state changes in child processes that have been spawned, e.g., a child process has died.
- Find out when input is available on file descriptors or output to a file descriptors has completed, e.g., using pipes.
- Receive notification of the expiration of an interval timer, e.g., so that periodic user feedback, such as a blinking caret, can be provided.
- Monitor, modify or extend the behavior of other notifier clients, e.g., noticing input events directed to a particular window client.

SunView is used in this paper as an example of an environment that uses the notifier. However, programs that are completely outside of the SunView context have utilized the notifier as well. The notifier is just a flow of control manager and can be used by any notification-based UNIX environment that has multiple clients. Because flow of control management is all that the notifier provides, the notifier serves as the common ground where disjoint packages can coexist with each other.

This paper describes enough of the programming interface to the notifier to impart the flavor of its use. The complete description of the interface is contained in the two SunView programming guides listed in the *References* section.^{3,4}

2. OVERVIEW

This section provides an overview of client interaction with the notifier.

A *client* of the notifier is any software entity that has *registered an event handler* with the notifier. A frame, a control panel, a scrollbar are examples of clients. The notifier uses a *client handle* as the unique identifier for a given client. The notifier does not interpret the client handle in any way, beyond requiring that it be unique. The most common way of generating a unique client handle is to use the address of an allocated data block that contains client specific data.

When a client starts up, it must advise the notifier of the types of events in which it is interested. A client does this by registering an event handler function for each type of event of interest. A client handle is used to identify the client to the notifier during registration. The bulk of event handler registration activity usually happens when a client is created, but may occur at any time thereafter.

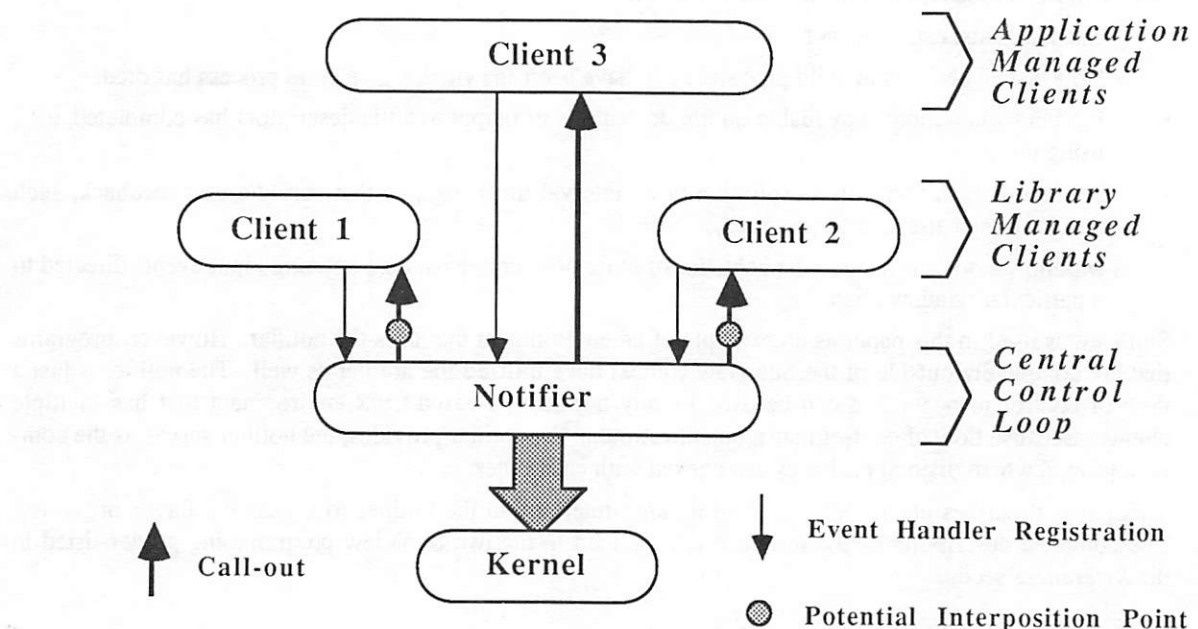
After clients have had a chance to register their event handlers, the application makes a call into the notifier to allow the notifier to detect events. When an event occurs, the notifier calls (dispatches to) the event handler appropriate to the type of event. In addition, the client handle is passed to each event handler when it is called, making it easy for each client to get to its instance specific data. When an event handler returns, control is returned to the notifier.

2.1. Types of Interaction

Client interaction with the notifier falls into the following functional areas:

- **Event handling** — A client may receive events and respond to them via event handlers. Event handlers do the bulk of the work in the notifier environment.
- **Interposition** — A client may request that the notifier install a special type of event handler (supplied by the client) to be inserted (or *interposed*) ahead of the current event handler for a given type of event and client. This allows clients to screen incoming events of other clients. Each such screened event may be redirected, discarded or simply monitored.
- **Notifier control** — A client may exercise control over when the notifier dispatches events. This is particularly important when porting programs into a notification-based environment like SunView.

The figure below shows an overview of the relationship of the notifier to its clients and the kernel.



3. EVENT HANDLING

This section describes how a client can be notified of UNIX-related events and client defined events. UNIX events are low-level occurrences that are meaningful at the level of the operating system. UNIX events include signals (software interrupts), input pending on a file descriptor, output completed on a file descriptor, tasks associated with managing child processes, and tasks associated with managing interval timers. Client defined events are, for example, the “a” key went down or a window has been uncovered.

A client establishes an interest in a certain type of event by registering an event handler procedure to respond to it. The client provides the event handlers to be registered. The event handler for a given type of event has a mandatory calling sequence. All event handlers return a value of either NOTIFY_DONE or NOTIFY_IGNORED depending on whether the event was acted on in some way or failed to provoke any action, respectively. This return value is useful when trying to extend a client’s functionality using interposition. If an event is ignored by a client’s event handler, the interposed event handler could feel free to provide some non-client supported interpretation of the event.

The procedure used to register an event handler returns a pointer to the function that was the previous event handler. The first time a client registers a given type of event handler, it will receive a pointer to a “no-op” function. The returned function pointer can be used to do explicit interposition, but the notifier supported interposition mechanism is easier to use.

The following sections describe the various types of events.

3.1. Child Process Control Events

Suppose that a client wants to fork a process to perform some processing on the client’s behalf. UNIX requires that some housekeeping of that process be performed. The minimum housekeeping required is to notice when that process dies and “reap” it. A client tells the notifier to call back whenever a child process changes state, e.g., dies, by registering a *wait3 event* handler via the call:

```
Notify_func
notify_set_wait3_func(client, wait3_func, pid)
    Notify_client client;
    Notify_func wait3_func;
    int pid;
```

In the above call, the *pid* identifies the particular child process that the client wants to wait for. The name *wait3 event* originates from the *wait3(2)* system call.

The reasons that the client doesn’t simply call *wait3(2)* are two fold. First, if *wait3(2)* was called and blocked, the thread of control would not be available to other clients in the user process. This situation should be avoided when there are other clients that are providing interactive user feedback. The second reason that an application shouldn’t call *wait3(2)* has to do with the semantics of the *wait3(2)* call. *wait3(2)* will return with status about *any* process that has changed state. Thus, if 2 clients are managing different child processes, they should hear only about their own process. The notifier keeps straight which client is managing which process.

3.1.1. Results from a process

An application that receives an exit code result back from a forked process should write a *wait3 event* handler. For example:


```

static Notify_value my_wait3_handler();
static int me, pid;

/* This is a code segment from main() */
pid = my_fork_and_exec();
/* Register a wait3 event handler */
(void) notify_set_wait3_func(&me, my_wait3_handler, pid);
/* Start dispatching events */
(void) notify_start();

static Notify_value
my_wait3_handler(me_ptr, pid, status, rusage)
    int *me_ptr;
    int pid;
    union wait *status;
    struct rusage *rusage;
{
    if (WIFEXITED(*status)) {
        /* Child process exited with return code */
        my_return_code_handler(me_ptr, status->w_retcode);
        /* Tell the notifier that I handled this event */
        return (NOTIFY_DONE);
    }
    /* Tell the notifier that I ignored this event */
    return (NOTIFY_IGNORED);
}

```

The call to `notify_start()` causes the program to go into the notifier's continuous central control loop. Note that the use of `&me` as a client handle is arbitrary, but illustrates one method of generating a unique client handle.

3.1.2. "Reaping" dead processes

Many clients using child processes simply need to perform the required reaping after a child process dies. These clients can use the predefined `notify_default_wait3()` as their wait3 event handler. Instead of a client explicitly removing a wait3 event handler, the notifier automatically removes a dead process's wait3 event handler from internal notifier data structures.

3.2. Input Pending Events

A program sometimes needs to know when there is input pending on a file descriptor. One common situation is waiting for input on one end of a pipe. A client can ask the notifier to call back whenever there is input pending on a file descriptor by registering an input pending event handler via the call:

```

Notify_func
notify_set_input_func(client, input_func, fd)
    Notify_client client;
    Notify_func input_func;
    int fd;

```

The notifier is used when waiting for input on a pipe when a program doesn't want to block on a `read(2)` or `system(2)` call. In the case of a SunView program, a program wants to avoid blocking so that the program can get back to the notifier's central control loop as fast as possible so that the user can interact with the window while waiting for input to become available on the pipe.

The file descriptor can be in blocking or non-blocking mode, or in asynchronous mode; the notifier handles both.

When any file descriptor is closed, one should *unregister* any event handlers associated with the now invalid file descriptor. This is done by passing in a `notify_func` of `NOTIFY_FUNC_NULL` to `notify_set_input_func()`. This method of passing in a `NOTIFY_FUNC_NULL` to unregister an event handler from the notifier works for any type of event.

3.3. Signal Events

Signals are UNIX software interrupts. The notifier multiplexes access to the UNIX signal mechanism. A client may ask to be notified that a UNIX signal occurred either when it is received (asynchronously) and/or later during normal processing (synchronously). The notifier can send both types of notifications.

Clients may define and register a signal event handler to respond to any UNIX signal desired. However, many of the signals that are caught by the application in a traditional UNIX program are caught by the notifier instead (see *Restrictions* below).

A client asks the notifier to call the specified function whenever a signal has been caught by registering a signal event handler via the call:

```
Notify_func
notify_set_signal_func(client, signal_func, signal, when)
    Notify_client client;
    Notify_func signal_func;
    int signal;
    Notify_signal_mode when;
```

`when` can be either `NOTIFY_SYNC` or `NOTIFY_ASYNC`. `NOTIFY_SYNC` means that a client wants to get the notification during normal processing, i.e., delay delivery of the signal so that the signal does not arrive when the program may be in the middle of doing something. Delivery is delayed until after control returns to the notifier. `NOTIFY_ASYNC` means that a client wants to get the notification when the signal is received, i.e., this mode mimics the old *signal(3)* semantics. A client can get called twice for the same signal if both a `NOTIFY_SYNC` and a `NOTIFY_ASYNC` signal event handler are registered for the same signal.

3.3.1. Asynchronous event handling

An asynchronous signal notification can come at any time (unless blocked using *sigblock(2)*). This means that the client can be executing code at any arbitrary place. Great care must be exercised during asynchronous processing.

It is rarely safe to do much of anything in response to an asynchronous signal. Usually, unless a program has taken steps to protect its data from asynchronous access, the only safe thing to do is to set a flag indicating that the signal has been received.

3.4. Timeout Events

A client may require notification of an expired timer based on real time (approximate elapsed wall clock time; `ITIMER_REAL`) or on process virtual time (CPU time used by this process; `ITIMER_VIRTUAL`). In order to receive this type of notification, the client must define and register a timeout event handler.

```
Notify_func
notify_set_itimer_func(client, itimer_func, which, value, ovalue)
    Notify_client client;
    Notify_func itimer_func;
    int which;
    struct itimerval *value, *ovalue;
```

The semantics of `which`, `value` and `ovalue` parallel the arguments to *setitimer(2)*. `which` is either `ITIMER_REAL` or `ITIMER_VIRTUAL`.

3.4.1. Polling

Interval timers can be used to set up a polling situation. There is a special value argument to `notify_set_itimer_func()` that tells the notifier to call back as often and as quickly as possible. This value is the address of the following constant:

```
struct itimerval NOTIFY_POLLING_ITIMER; /*{{0,1},{0,1}}*/
```

This kind of high speed polling can hog all of a machine's available CPU time, but may be appropriate for high speed animation.

3.4.2. Turning the interval timer off

Specifying an interval timer with no time value with which to reset the timer upon expiration causes the notifier to flush any knowledge of the interval timer after it delivers the timeout event. Otherwise, supplying a NULL interval timer pointer to `notify_set_itimer_func()` immediately removes the client's interval timer from the notifier.

3.5. Client Events

This section describes how *client events* are handled by the notifier. Clients define and originate client events, the notifier simply delivers them to the appropriate recipients. The notifier is responsible for dispatching client events to a client's event handler after the event has been *posted* to the notifier by client code. To register a client event handler call:

```
Notify_func  
notify_set_event_func(client, event_func, when)  
    Notify_client client;  
    Notify_func event_func;  
    Notify_event_type when;
```

`when` indicates whether the event handler will accept notifications only when it is safe (NOTIFY_SAFE) or at less restrictive times (NOTIFY_IMMEDIATE).

3.5.1. Posting

A client event may be posted with the notifier at any time. The poster of a client event may suggest to the notifier when it is to be delivered to the client, but this is only a hint. The notifier will see to it that it is delivered at an appropriate time. Normally, the notifier sends client event notifications when it is *safe* to do so. This may involve some delay between when an event is posted and when it is delivered. In particular, the notifier delays delivery if a client is in the middle of handling another event. However, a client may ask to always be notified of the posting of a client event right when it happens, i.e., *immediately*.

The immediate client event notification mechanism should be viewed as an extension of the UNIX signalling mechanism in which events are client defined signals.

When SunView posts a client event, it posts a pointer to a fixed field structure. There is sometimes a need to pass additional data with an event. An example of this is when a scrollbar posts an event to its owner to do a scroll. The notifier allows the scrollbar's client handle to be passed as an argument, along with the event, when posting the client event.

4. INTERPOSITION

The notifier provides a mechanism with which an application can intercept calls out to event handlers. This mechanism is called *interposition* and is a powerful way to both monitor and modify client behavior in ways that extend the functionality of a client.

Interposition allows a client to intercept an event before it reaches the *base event handler*. The base event handler is the one originally set by a client. Clients may use interposition to monitor and filter events coming into an event handler.

4.1. How interposition works

A client requests that the notifier install an interposer function, supplied by the client, for a specified client and type of event. When an event arrives, the notifier calls the function at the top of the interposition list for that client and that type of event. An interposed routine may (indirectly) call the next function in the interposition sequence and receive its results.

4.2. Uses of interposition

Typically, it is application level code that uses interposition. But, in general, any client's creator may want to use interposition. There are many reasons why an application might want to interpose a function in the call path to a client's event handler.

- An application may want to use the fact that a client has received a particular event as a trigger for some application specific processing.
- An application may want to filter the events to a client, thus modifying the client's behavior.
- An application may want to extend the functionality of a client by handling events that the client is not programmed to handle.

SunView window objects utilize the notifier for much of their communication and cooperation. Thus, if an application wanted to monitor the user actions directed to a particular window then the application would use interposition to get into the flow of control.

4.3. Interface to interposition

The notifier supports interposition by keeping track of how interposition functions are ordered for each type of event for each client. Here is a typical example of interposition:

- An application creates a client. The client has set up its own client event handler using `notify_set_event_func()`.
- The application tells the notifier that it wants to interpose its function in front of the client's event handler by calling `notify_interpose_event_func()` (uses the same calling sequence as `notify_set_event_func()`).
- When the application's interposed function is called, it tells the notifier to call the next function, i.e., the client's function, via a call to `notify_next_event_func()` (uses the same calling sequence as that passed to the interposer function, not described here).

5. NOTIFIER CONTROL

When an application is using a notification-based programming style, the notifier's central control loop is explicitly entered via a call to `notify_start()`. To break out of the central control loop `notify_stop()` is called. Also, the central control loop is exited if the notifier no longer has any clients registered.

5.1. Porting Programs to SunView

Most programs that are ported to SunView are not notification-based. They are traditional programs that maintain strict control over their main control loop. Much of the state of such programs is preserved on the program stack in the form of local variables. Converting such programs to be notification-based can be tedious. The notifier supports the non-notification-based form of programming so that an application can use SunView packages without inverting the control structure of its program to be notification-based.

5.1.1. Explicit dispatching

The simplest way to convert a program to coexist with the notifier is called *explicit dispatching*. This approach replaces the call to `notify_start()`, which usually doesn't return until the application terminates, with a call to `notify_dispatch()`. `notify_dispatch()` goes once around the notifier's central loop, dispenses any pending events, and returns. The application is responsible for calling

`notify_dispatch()` often enough so that an adequate event flow to notifier clients can be maintained. Explicit dispatching is good when an application is doing some computationally intensive processing and the application wants to occasionally give the user a chance to interact with SunView.

5.1.2. Implicit dispatching

There is another method of interacting with the notifier that is useful when an application simply wants the notifier to take care of its clients and block until there is something of interest to the application. This is called *implicit dispatching*.

This time, one replaces the call to `notify_start()` with the following bit of code:

```
/* Enable implicit dispatching */
(void) notify_do_dispatch();
while (!my_done) {
    char c;
    ...
    /* read allows implicit dispatching by notifier */
    if ((n = read(0/*stdin*/, &c, 1)) < 0)
        perror("my_program");
    ...
}
```

`notify_do_dispatch()` allows the notifier to dispatch events from within the calls to `read(2)` or `select(2)`. The notifier's version of `read(2)` and `select(2)` won't return until the normal versions would. They can block exactly like the normal versions.

`notify_no_dispatch()` prevents the notifier from dispatching events from within the call to `read(2)` or `select(2)`. This is the default state.

5.2. Prioritization

Although not normally the case, a client might have multiple events pending for distribution at once. For example, input pending on a file descriptor and a timer expired. The order in which a particular client's events are dispatched may be controlled by providing a *prioritizer* function. Most clients rely on the ordering provided by the default prioritizer.

5.3. Scheduling

There is a mechanism for controlling the order in which multiple clients are notified. Controlling the order in which anyone particular client's notifications are sent is done by that client's prioritizer function. Replacement of the default scheduler will be done most often by a client that wants to run ahead of other clients. For example, if doing "real-time" cursor tracking in a user process, the tracking client would want to schedule itself ahead of other clients.

6. RESTRICTIONS

The notifier imposes some restrictions on its clients. Designers should be aware of these restrictions when developing software to work in the notifier environment. These restrictions exist so that the application and the notifier don't trip over each other. More precisely, since the notifier is multiplexing access to user process resources, the application needs to respect this effort so as to not violate the sharing mechanism.

For example, a client shouldn't call `signal(3)`. The notifier is catching signals on the behalf of its clients. If a client sets up its own signal handler over the one that the notifier has set up then the notifier will never notice the signal. One should call `notify_set_signal_func()` instead of `signal(3)`. Other calls in the same restricted class include `sigvec(2)`, `setitimer(2)`, `alarm(3)`, `getitimer(2)`, `wait3(2)`, `wait(2)`, `ioctl(2)(...,FIONBIO,...)`, and `ioctl(2)(...,FIOASYNC,...)`.

As another example, a client shouldn't be catching `SIGALRM` as part of the process of detecting an interval timer expiration. The notifier call `notify_set_itimer_func()` takes responsibility for setting up a

SIGALARM signal catcher as well as calling *setitimer*(2). Other signals in the same semi-restricted class as SIGALRM are SIGVTALRM, SIGTERM, SIGCHLD, SIGIO and SIGURG.

7. IMPLEMENTATION

Describing the programming interface to the notifier is the main point of this paper. However, a brief description of the implementation of the notifier is included in this section.

The notifier has three major internal components: the *detector*, the *dispatcher* and the *interposer*. The detector maintains a list of clients that have event handlers registered. Each client has its own list of event handlers. As the notifier's event handler list is modified, the notifier recomputes its knowledge of which events are now of interest to its clients and positions itself to detect the occurrence of those events. Thus, signal handlers are set, interval timers are set, input/output/exception file descriptors bit masks are computed. When control is passed to the notifier for the purpose of dispatching events, the notifier goes at least once around its central control loop. This involves making sure that the file descriptor bit masks are up-to-date, (usually) making a *select*(2) system call, figuring out what events have occurred and enqueueing an event handler call-out with the dispatcher. The dispatcher is then called to deliver the events. When the dispatcher is through, the notifier either returns back to the application or goes around the central control loop again.

The dispatcher goes through a process of figuring out the order of notifying clients. Once a client is scheduled, the dispatcher goes through a process of figuring out the order of notifying event handlers for a single client. Both of these ordering operations may be controlled by clients by them providing alternative functions to call instead of the default functions.

When actually calling out to an event handler, the interposer controls the sequencing of calling the functions interposed before a base event handler.

The notifier takes pains to protect itself from the randomness of asynchronous calls into itself. For example, the notifier maintains and protects its own internal storage pool separate from the heap so as to not be in a position to corrupt the heap through asynchronous access.

The notifier also does everything it can to minimize its CPU overhead. This is done by caching as much state as possible so as to avoid searching the client and event lists.

The notifier replaces the *read*(2), *select*(2) and *fcntl*(2) system calls with library routines of its own. The generic *syscall*(2) is used to actually make these calls into the kernel. The functionality of the *select*(2) system call is relied upon to minimize the trips to the kernel. However, a system V version of the notifier could be made to work, albeit with greater overhead, without changing the interface to the notifier.

8. ISSUES

Here are some issues surrounding the notifier:

- Not all process resources are multiplexed (e.g., *rlimit*(2), *setjmp*(2), *umask*(2), *setquota*(2), *setpriority*(2)), only ones that have to do with flow of control multiplexing. Thus, some level of cooperation and understanding need exist between packages in the single process. Careful and limited use of *setjmp*(2) is particularly important.
- The notifier is not a lightweight process mechanism that maintains a stack per thread of control. However, as such a mechanism becomes available, the notifier will still be valuable for its support of notification-based clients. The notifier could coexist with a lightweight process mechanism; their dual usage should not be mutually exclusive.
- Client events are disjoint from UNIX events. This is done to give complete freedom to clients as to how events are defined. One could imagine certain clients wanting to unify client and UNIX events. This could be done with a layer of software on top of the notifier. A client could define events as pointers to structures that contain event codes and event specific arguments. The event codes would include the equivalents of UNIX event notifications. The event specific arguments would contain, for example, the file descriptor of an input pending notification. When an input pending notification from the notifier was sent to a client, the client would turn around and post the equivalent client event notification.

- The layer over the UNIX signal mechanism is not complete. Signal blocking (*sigblock*(2)) can still safely be done in the flow of control of a program to protect critical portions of code as long as the previous signal mask is restored before getting back to the notifier. Signal pausing (*sigpause*(2)) is essentially done by the notifier. Signal masking (*sigsetmask*(2)) can be accomplished via multiple *notify_set_signal_func*() calls. Setting up a process signal stack (*sigstack*(2)) can still be done, the notifier doesn't care. Setting the signal catcher mask and on-signal-stack flag (*sigvec*(2)) could be done by reaching around the notifier but is not supported.

Here are some issues that reflect on the current implementation of the notifier and may be addressed in the future:

- The notifier's central control loop is not reentrant. It would be useful to be able to recursively call *notify_start*() so that a given point in a program could temporarily preserve its state on the stack while at the same time allowing event notification to continue.
- The library functions *close*(2) and *dup*(2) could be intercepted so that the notifier is not waiting on invalid or incorrect file descriptor's if a client forgets to remove its conditions from the notifier before making these calls.
- The library functions *signal*(3) and *sigvec*(2) could be intercepted so that the notifier doesn't get fouled up by programs that fail to use the notifier to manage its signals.
- The library function *setitimer*(2) could be intercepted so that the notifier doesn't get fouled up by programs that fail to use the notifier to manage interval timers.
- The library function *ioctl*(2) could be intercepted so that the notifier doesn't get fouled up by programs that use FIONBIO & FIOASYNC instead of the equivalent *fcntl*(2) calls.
- The library functions *readv*(2), *writew*(2) and *write*(2) could be intercepted just like *read*(2) and *select*(2) so that a program doesn't tie up the process.

9. SUMMARY

The notifier has proved very useful in program environments that contain a variety of relatively disjoint clients that need separate access to the flow of control within a user process. Rich applications have been constructed without complex control flow code at the application level.

References

1. Sun Microsystems Inc., *Windows and Window Based Tools: Beginner's Guide*, Revision A of 16 February 1986 (Part No: 800-1287-03).
2. Kepecs, Jonathan, "Lightweight Processes for UNIX: Implementation and Applications," in *USENIX Conference Proceedings*, Summer 1985.
3. Sun Microsystems Inc., "The Notifier," in *SunView Programmer's Guide*, Revision A of 16 February 1986 (Part No: 800-1345-02).
4. Sun Microsystems Inc., "Advanced Notifier Usage," in *SunView System Programmer's Guide*, Revision A of 16 February 1986 (Part No: 800-1342-02).
5. Sun Microsystems Inc., "wait(2)," in *UNIX Interface Reference Manual*, Revision G of 17 February 1986 (Part No: 800-1303-02).

Error Recovery in a Stateful Remote Filesystem

*Alan Atlas
Perry Flinn*

MASSCOMP[†]
One Technology Park
Westford, MA 01886

ABSTRACT

Our experiences in designing and implementing a remote filesystem using a stateful approach are described. A stateful approach, which is required in order to strictly maintain UNIX[‡] filesystem semantics and behavior, can successfully be used, but it does require close attention to error conditions. Error avoidance, detection, and recovery are major areas of effort during implementation and are described in detail.

State information of interest includes reference counts on inodes, current network transaction information, current remote mounts (on a client and on a server), and resource locks (inodes, devices, etc.).

User processes on the client accomplish both detection and correction of errors. A more global approach is necessary on the server where some number of kernel processes are engaged in unknown activities.

Error avoidance is based on limiting the amount of state information by careful design. Many system calls can be handled without leaving residual state.

Error recovery schemes are described which prevent the failure of any client or server due to the failure of any other. As long as it is possible to detect a failed network transaction, it is possible to arrange to clean up any information or processes left on a client or server which would otherwise remain.

1. Introduction

Network filesystems are now a reality in the UNIX world. Along one particular dimension, they may be categorized as either "stateless" or "stateful". The two strategies are distinguished by one aspect of the client-server relationship: whether or not the history of the transactions between a client and a server affects the execution of the next transaction. The stateless approach does not depend on history and has the advantage of simplicity. The stateful approach does depend on the history of the transactions. The stateful approach can lead to a remote filesystem which maintains UNIX filesystem behavior across the network, but it has the drawback that recovery in the

[†]MASSCOMP and RTU are Trademarks of Massachusetts Computer Corporation

[‡]UNIX is a Trademark of AT&T Bell Laboratories

face of network or host failures can be difficult. However, this need not have an impact on the average user so long as the implementation deals adequately with the error recovery problems. This paper documents our experiences during the design and coding of the Extended File System (EFS) at MASSCOMP in terms of the error recovery strategies we developed. Its purpose is to report the strategies which we found to be successful and to discuss extensions of the error recovery system which we are currently investigating.

2. Design Overview

The basic design of the Extended File System for MASSCOMP's Real Time UNIX operating system (RTU) was reported at the Summer 1985 USENIX Conference. This section presents a brief overview, focusing on those aspects that relate to error recovery.

2.1. Requirements

The fundamental requirements which guided the design of EFS were:

1. Existing UNIX filesystem semantics had to be maintained for accesses to remote files.
2. The impact of integrating EFS with other (then) current kernel development work had to be minimized.
3. EFS had to be configurable in the kernel so that customers not requiring its functionality could omit it.
4. Existing programs had to be able to access remote files without recompilation.

The overriding design requirement for EFS was the first of these—that all UNIX semantics be preserved. By implication, speed was of less importance than filesystem integrity. Further, because complexity of integration into the kernel was to be minimal, EFS was consciously kept from greatly changing the main kernel data structures. It was the necessity to preserve UNIX semantics which drove the stateful design.

Preserving UNIX semantics puts a formidable burden on remote filesystem design. The traditional UNIX filesystem behaves as it does based on the assumption that it is running on an isolated machine with locally attached mass storage. Its behavior during contention for files and filesystem resources is well known. In order to duplicate this behavior across hosts separated by a network, and to least intrusively incorporate EFS into the existing and changing RTU kernel, we chose the approach of extending filesystem semantics across the network rather than changing the filesystem to suit the network. This stateful approach has achieved the desired goals within the given constraints.

2.2. Connected Resources

Central to error handling concerns within EFS is the idea of a connected resource. A connected resource (e.g., an inode) is one which is being manipulated by a collection of hosts made up of some number of clients and the server on which the resource physically resides. In our design, file resources remain centralized on the server. Clients using a remote file retain a reference (called a *remote inode*) to the actual inode on the server. Client operations on the remote inode such as *iput()* or *readi()* cause related operations to be performed on the actual inode on the server.

Connected resources are the source of the majority of error recovery problems. If a transport failure causes either side of a connected resource to be abandoned in some way, the other side must detect the problem and adjust its local state so that client and server both agree on the state of the connected resources between them. For example, the reference count on a server's inode reflects the total number of references to the inode within the EFS net. Clients cannot merely change the count of a remote inode, they must also notify the server where the corresponding physical inode exists so that the in core reference count of that inode can be adjusted.

2.3. Architecture

EFS is based on the client-server relationship between two hosts established with the *remote mount* operation. The client initiates most activity; the server provides requested services. Every RTU kernel that is configured with EFS incorporates both client and server functionality although the two halves operate mostly independently of each other.

2.3.1. Remote inodes

A client host performs a remote mount operation to establish the mapping from a local inode to one on a server. This operation puts a remote inode into the client's mount table where the root inode of a disk-resident file system would normally be found. Remote inodes are known to contain only enough information to allow the real inode on the server to be accessed. Actual inode contents (owner, times, etc.) and file data are retrieved from the server when needed. EFS ensures that for every remote inode in a client's inode table, the corresponding real inode is maintained in core on the server. In essence, a remote inode on a client is guaranteed to correspond to a reference count on a real inode on the server. A server inode's reference count is the sum of the reference counts on all corresponding remote inodes on client hosts, plus the number of local references to the inode. This means that the appearance of an inode in core on the server, its active life there, and its eventual disappearance can be controlled entirely in response to a client host's wishes.

2.3.2. Name Lookup

Name lookup is currently done in the traditional manner of parsing pathnames and retrieving inodes. When a remote inode is encountered during a pathname walk, the unparsed portion of the pathname is packaged into an EFS request message and sent to the appropriate server. Included in the message with the pathname remnant is a pointer into the server's in-core inode table for the directory inode from which the path search is to continue. This pointer is the result of a previous *rmount* or *chdir* operation, and is retained in the remote inode along with the server's network address. Depending on the purpose of the name lookup, a system call may be completed at this point without the creation of a new remote inode on the client. However, if the operation is an *open*, for example, a new remote inode will be needed on the client for subsequent file operations, so the reference count of the server's in-core inode is incremented and a pointer to it is returned to the client.

2.3.3. Client Operation

In general, each system call that could involve a remote file has been modified to determine whether this is the case and if so to call an appropriate client routine:


```

chmod()
{
    if (REMOTE_INODE) {
        chmod_client();
        return;
    }
}

```

The client routine sends one or more request packets to the server, where the system call is executed by an agent process. Error codes or data are returned to the client where necessary, and transport failures are detected and handled by the client (user) process.

2.3.4. Server Operation

The server side of an EFS host consists of a pool of kernel processes which exist forever to do work on behalf of clients. The pool is of (configurable) fixed size except for certain operations which require that a new agent process be created for the duration of the operation. The *efsdaemon* receives new transaction requests from the network and puts them in a queue of work which is serviced by *efs_agent* processes.

A new *efs_agent* process is created for each new *lockf()* by a client and for requests to operate on special files which may cause indefinite sleeps. Resources which may cause very long sleeps (such as reading a tty) require the services of a temporary *efs_agent* to avoid using up all of the available agents on a particular server. File locks require a dedicated agent for the duration of their existence.

2.4. Transport Mechanism

EFS uses a specially designed transport protocol (RDP) that provides reliable delivery of datagrams. RDP incorporates a mechanism for dynamically establishing sequences of related datagrams, known as transactions, and for maintaining many simultaneous transactions using a single socket. This multiplexed transaction capability serves several purposes:

1. It supports the request/response communication structure that is characteristic of most EFS client-server interactions.
2. It simplifies the implementation of multi-packet exchanges that occur, for example, during the processing of large *read* and *write* operations.
3. It provides a key element in the detection of failures by returning specific error indications when a datagram cannot be successfully sent after several retries, or when a transaction is broken because one of the peers has failed to elicit a response from the other for a period of time.

RDP provides error indications as return values from *send* and *receive* operations. As a general rule, most such errors cause the related transaction to be aborted. The exception is the ENOBUFS error, which arises when the local system is unable to allocate sufficient memory to send a new datagram. This allows the client or server process the option of retrying the operation after a delay, on the assumption that sufficient memory space will have been released during the interim. This approach has proven particularly valuable in our systems that use a network coprocessor, which has very limited memory resources and is often subject to transient shortages during periods of heavy load. RDP is designed to ensure that if one end of a transaction encounters a fatal (to the transaction) error during a *send* or *receive* operation, the other end will

also be given a corresponding error indication within no more than a few seconds.

2.4.1. More on Transactions

An RDP transaction is created as a side effect of a *send* operation under control of two special flags: `SF_MORE` and `SF_REPLY`. If `SF_MORE` is specified, further datagrams may be sent on the same transaction. If `SF_REPLY` is set, the receiver of the first datagram may send one or more replies to the originator. Thus, the sender of the opening datagram controls whether the transaction is duplex or simplex. Once the transaction is established, each direction of transfer is controlled by the peer sending in that direction, and remains viable until a datagram is sent *without* the `SF_MORE` flag, or a fatal transport error occurs. There is no other mechanism for “closing” a transaction. This results in some minor complications in the handling of non-transport-related errors. For example, if a filesystem error occurs during a multi-block *write*, the server must continue to consume datagrams from the transaction until it receives one without the `SF_MORE` flag. If this were not done, the transaction would linger on, unnecessarily consuming already scarce memory resources.

2.4.2. Network Access as a Limited Resource

MASSCOMP supports a family of machines which cover a broad range of hardware. Lower end machines use an in-kernel network implementation, while the other machines use a separate coprocessor to provide Ethernet access. Buffer space on the coprocessor is limited, as is computational power. We have found that it is possible to require more from the communications hardware than it is prepared to give. This can lead to deadlock situations. For example, an agent process holding a lock on some resource may block trying to send on the network because of a memory shortage on the coprocessor. If this shortage occurs because the controller is holding data not yet received by other agent processes which are themselves blocked waiting for access to the resource held by the first agent, a deadlock occurs. By keeping an active agent attached to all locked resources on the server and carefully coordinating the system call protocols to avoid tying up network resources, we have been able to virtually eliminate this problem.

3. Statefulness

In a stateful remote filesystem, the state of the relationship between two hosts can largely be described by the connected resources shared by the two machines. Information about the state of the relationship is kept on both the client and the server so that in the event of a transport failure, the two hosts can independently reestablish their relationship. This may be simply agreement that the entire relationship has been severed, or it may involve adjusting one side's view of a connected resource so that it agrees with the other side's view.

The success or failure of an EFS request in part depends on the fact that the state of the server/client relationship is known to both. For example, if a process' current directory is remote, name searches start on the server using a pointer into the server's inode table supplied by the client. Clearly, the history of the relationship must be agreed upon for such a close coupling to work.

Since reference counts on inodes must balance between server and clients, an *iput()* of a remote inode not only decrements the count on the remote inode on the client, but also sends a packet to the server. The corresponding inode on the server is *iput* there and the proper count is maintained. Occasionally, the *send* operation may fail causing

a rogue reference count to be left on an inode in core on the server. If the failure were ignored, the inode table on the server would eventually fill up. Worse, the failure may result from a transient network problem; we don't want to remove the connection or take drastic measures which impede EFS's ability to continue to work through network glitches. A mechanism to retry the failed send at some later time solved this, but it required that one of the server side agents do the work.

Maintaining state information in an error-free environment is not difficult. The problems arise when errors begin to occur either in the transport mechanism or in the normal course of kernel execution. Detecting errors and purging extraneous or incorrect state information in a transparent way then becomes a challenge which must be overcome if a useful and robust remote filesystem is ever to be realized.

3.1. Degrees of State

File-related system calls fall into two categories, distinguished by the amount of state information required to implement them. Functions in the so-called "stateless" category accomplish their operations by sending a single request message and receiving a single response. The request typically includes a pathname which is valid on the server and any arguments necessary to the call. Once on the server, the entire call is executed from name lookup to completion. Execution of one of these functions requires no state information on the server other than the previously established remote mount, and creates no additional state following its completion. Examples in this category are *chown(2)*, *chmod(2)* and *unlink(2)*.

A variation on this category is the set of functions that require multiple exchanges between client and server, but still accomplish their purpose in the context of a single RDP transaction and leave no residual state on the server. Examples of these functions are *read(2)*, *write(2)*, *rename(2)*, and *link(2)*.

Functions in the "stateful" category depend on setting up a connected resource once and taking advantage of its existence for subsequent access. This means that even when the client is not actually causing activity on the server there are resources there that are under the control of the client. These calls are *chdir(2)*, *chroot(2)*, *open(2)*, *creat(2)* and *rmount(2)*.

3.2. State and Connected Resources

3.2.1. Remote Inodes

Remote inodes are central to EFS. The creation of a remote inode on a client involves getting the real inode on the server into core or adding a reference count to one that is already there, returning a pointer to that inode to the client, and putting the pointer and the server's internet address into a remote inode which is in the client's inode table. Managing reference counts between remote inodes and real inodes isn't too difficult. Each time an inode pointer is exported to a client, the server keeps a record of the pointer and the client responsible for the reference count and the number of reference counts that client is responsible for. If the client becomes unreachable later on, the server need only consult this record and *iput* the appropriate inodes the correct number of times.

There is one perverse case worth mentioning which has proved to be very rare but difficult to completely handle. During an *exec*, the reference count on the inode of the current directory of the process doing the *exec* is incremented. If the directory is a remote inode, it is possible that the attempt to increment the count on the server will

fail. If the transport failure is permanent then the server will eventually sense the absent client and clean up all of that client's state. If the failure is transient, then the server will simply not know that its reference count on the real inode is wrong. This failure will not abort the *exec*, but when *exit()* tries to *iput* the remote inode, it must not *iput* on the server since the count was never incremented there. This is accomplished by invalidating the remote inode in such a way that the *send* during the *iput* will fail but the failure will not be recorded by the client for later retries.

3.2.2. Remote Locks

Because coprocessor network implementations can be too limited in terms of simultaneous access by multitudes of clients, we have adopted a general rule that no connected resource be left locked on a server by a client without an agent process executing code which manages that resource. In the event of network failure, the agent process can remove the lock and clean up quite easily. File locks using *lockf(2)* are therefore established with a temporary *efs_agent* as caretaker for the duration of the lock. This process does nothing other than the locking and unlocking operations. The rest of the time it sleeps on a *receive* waiting for traffic from the client which established the lock. The client can get to the caretaker process through an extension to the file table entry on the client, and other EFS activity can be accomplished without using the caretaker for anything other than the locking/unlocking. The *receive* operation to the network will return an error to the caretaker process if the client fails to maintain an open connection, so the lock will always be removed in case of transport failure.

Because all lock administration is done at the server, deadlock searches can easily be accomplished.

3.2.3. Special files

The pool of kernel processes on the server which execute on behalf of clients is a limited resource. When dealing with special files, particularly ttys, the possibility of an agent sleeping forever precludes allocating one of the few *efs_agents* to a read of a tty. As with file locking, a temporary agent is created which deals with accesses of special files. Because the agent can be asleep on the device in question rather than the network, it is possible for the network to fail without the agent's knowledge. These agents can be identified by their proc table entries, and can be awakened by the *efsdaemon* in such a way that they will sense the network failure and clean up their device before exiting.

4. Error Recovery Strategy

The error recovery scheme we have developed is based on keeping a record of all connected resources and providing a mechanism whereby an agent process can be used to correctly adjust for the error. Hosts can be unable to communicate for a number of reasons: failure of the communications coprocessor, failure of the host itself, or transient or permanent network transport problems. Once communication between client and server is interrupted, the two may not agree on the state of their relationship.

The general strategy for dealing with errors revolves around data structures for maintaining state recovery information and the use of agent processes to do cleanup work. The server actively does most of the error detection work.

4.1. Error Recovery vs. State Restoration

Once an error has occurred and the decision to take remedial action has been made, there remains the question of what the goal of remedial action should be. Consider the case of asynchrony between client and server. If the client thinks there is a remote mount and the server thinks there isn't, should the server reestablish one or should the client clean up its mount table and inode table? The question is one of recovery versus restoration. Assuming either action is possible, the two hosts must reach agreement on the state of their relationship. Either both will know about the same remote mount, or both will know that there isn't one. Once agreement has been reached, things can continue.

EFS currently adopts a recovery attitude. That is, in the face of errors requiring recovery action, both client and server return to the most easily reached state independently. This usually means that rather than rebuild the state of a particular transaction, EFS clears references to the affected resources. This often causes subsequent calls to fail, but it does insure that both machines continue to function properly in the face of network error.

The idea of rebuilding the state of a transaction so that, for example, a multi-block *write* to a file would be able to continue through a crash and reboot of a server automatically is being investigated. We believe we have the information necessary to restore state, assuming the client wants to do that. Restoring the state of some transaction would involve automatically reestablishing the appropriate remote mount, opening any remote files which were open, setting up current directories and other inodes which were in core on the server, and putting back locks and special file agents. The client is in a good position to do this, but the amount of work would be large.

4.2. Server Error Procedures

4.2.1. Data Structures for Error Recovery

The server keeps track of remote mounts, inodes which have counterparts on clients, and locks which have counterparts on clients. The *rmount* table maintains a record of clients with remote mounts established to the server. Its role is to aid in detection of asynchrony and dead clients.

A hashed array, called the *rtbl*, is used to keep track of inodes which are part of connected resources. There is a separate entry made in the table for every resource/client pair. There would be two entries in the table if two different clients created remote inodes for the same server inode. In this way, if one client fails, its reference count on the server's inode can be adjusted without affecting the count for the another client.

There are no tables for file locks or special file accesses since an agent process is always in control of them and possesses the state knowledge to correctly manage them in the face of network or client failure. Some difficulty occurs in synchronizing recovery activity around a heavily used file with many locks, readers, and writers, but it can be sorted out.

4.2.2. Asynchrony

Asynchrony refers to a situation wherein a client and a server disagree on which remote mounts are current at a certain time. Occasionally a server can crash and reboot without the client's actually sending it any requests during that time. A rebooted server will of course have no knowledge of the remote mounts which had been

set up before it crashed, but the client won't know that it has crashed and will have no reason to suspect that the remote mounts it has are no longer valid.

Error detection begins when a request for a new EFS operation arrives at the *efs-daemon* on the server. The *rmount* table is checked for the client that sent the request. If the client appears in at least one entry, it means that at least one remote mount is current from that client. The time of the request is entered in the *rmount* table for later use. If there is no *rmount* table entry for that client and the request is not itself an *rmount*, then something has happened to get client and server out of step.

A further check is supplied by the `FIRST_RMOUNT` and `LAST_RUMOUNT` flags, which are sent by the client as part of the remote mount protocol. The `FIRST_RMOUNT` flag is interpreted as meaning there should be no record of this client on this server until this remote mount is established. `LAST_RUMOUNT` means that after the current unmount request is completed, there should be no references to the current client on this server. These provide a global check for abandoned resources that may not have been properly cleaned up. The server can quickly ascertain whether or not it agrees with the argument that it should have no connected resources to that client and take action if necessary.

Massive network failure such as that caused by downloading the coprocessor or losing the tap on the Ethernet cable is detected by the *efsdaemon*, which receives an `ENETDOWN` error from the network. This causes all client information to be purged. Inode counts are adjusted downward using the contents of the *rtbl*, special file and lock agents are awakened, the *rmount* table is cleared, as is the request queue. When transport is working again, any clients which were not disturbed during the failure will receive an `ENORMOUNT` error if they try to access the server through old remote inodes.

4.2.3. Unresponsive Clients

The time field in the *rmount* table is used to establish the fact of recent contact with a client. At regular time intervals, an EFS server puts a request into its own work queue to be executed by the next available agent process. This request specifies that all clients mentioned in the *rmount* table which haven't been in contact since the last time this "pinging" was done must be interrogated concerning their health.

The inquiry is sent as a new EFS transaction, so the *efsdaemon* on each client actually fields the request and returns an answer. If no useful response is forthcoming from a client, it is declared dead.

4.2.4. Interrupted Transactions

RDP provides a means whereby the success or failure of a network operation is immediately known by the requester. This is not the result of an EFS transaction, but rather the result of an individual *send* or *receive* call. On the server, a negative (error) return from a network call implies that the client will be abandoning this transaction, since the error propagates to the other end of an RDP connection. Usually, the agent simply exits when this happens. Only the pinging of unresponsive clients discussed above is used to establish an unreachable client. Agents executing normal server routines assume the network error is due to a temporary condition. This means that servers of the stateful calls do not clean up state information in this instance. The client will be trying to do that. If it succeeds, fine, if not, it will be because it had massive failure.

4.2.5. Server Error Recovery

Once an error condition is detected, action is taken to realign the client and server. For asynchrony, the server removes any traces of the affected client. This means looking through the *rtbl* array and adjusting any counts on inodes that are on record as being for the client, removing any entries in the *rmount* table for the client, and removing any locks or hung agent processes due to the client. At that point, both client and server agree that there is no connection established. Once that has happened, the next request for a remote mount can be safely honored.

Inability to ping a client is taken to mean that the client is dead. This decision is sensitive to the timeouts used in trying to send, since the communications coprocessor suffers in performance when heavily loaded. Once the decision is made, the server returns to an easily identifiable state of having no connection to the dead client. Any entries in the *rtbl* attributable to the dead client are cause for adjusting inode counts. Inodes which are connected to clients are never left locked in between being accessed by an *efs_agent* process.

RDP transactions which are in progress when a host dies notify any callers which are blocked when the failure occurs. This enables agent lock processes and device processes to be awakened and perform their own recovery procedures. The approach is to close out the resource gracefully so as not to prevent other activity.

4.3. Client Error Procedures

4.3.1. Data Structures for Error Recovery

The client side of EFS generally uses kernel data structures which are already in place. Remote mounts are recorded in the mount table; executable files are managed through the text table; remote inodes are found in the inode table. The one special data structure on the client side is a table of failed attempts to perform resource management on a connected resource. The failure of a *send* operation during an attempt by a client to *iput()* an inode on a server is recorded here. These failed attempts are retried until they either succeed or EFS is reset in some way.

4.3.2. Asynchrony

Before each *rmount(2)* and *rumount(2)*, the client checks its mount table to determine how many remote mounts already exist to the given server. The *FIRST_RMOUNT* flag is set in an *rmount* request to a server not already represented in the mount table. This provides a means whereby the server can verify that it agrees with the client that there is no existing remote mount between them. If there is disagreement, the server recovers.

If an *rumount* operation will remove a server entirely from the client's mount table, this fact is communicated to the server with the *LAST_RUMOUNT* flag. Again, if the server disagrees, it takes corrective action.

If an indication is returned at any time from the server that it has no current remote mount for this client, then the client process which receives that information cleans out its inode and mount tables by invalidating remote inodes for that server and removing the mount table entry. The remote inodes eventually disappear as they are freed by the processes owning reference counts. No impact is felt on the server, however, since the invalidated remote inode information causes any attempts to send using that remote inode to fail.

4.3.3. Unresponsive Server

Transient failures communicating with servers are ignored by clients. Sometimes it is possible to retry a particular system call, which the client calls do automatically. If not, the system call returns an error.

4.3.4. Client Error Recovery

Client processes make use of the mount table and the inode table as a matter of course. Information which EFS uses is integrated into both of these tables in the form of remote inodes in the inode table and an MREMOTE flag on entries in the mount table. In general, the client's vulnerability to state errors is less than the server's. It always has the ability to clear out references to connected resources whether or not its actions can be communicated to the server. Even if the server was not attached to the net, processes on the client would be able to *iput* remote inodes, unmount remote directories, and remove remote locks.

Of course these operations would affect only the client's idea of the state of remote resources with respect to that server. Transient network problems complicate this picture. If Client A tried to *iput* a remote inode, it could purge its remote inode from core but be left with a failed *send* call and the certain knowledge that if it is alive, Server B now has an abandoned resource (the reference count on the real inode) on its hands. If Server B was alive but its network had stopped working for a few moments, it would have no way of detecting the fact that it now had an abandoned resource in core.

This situation requires the client to keep its own state recovery table. The contents of this table are entries for failed attempts to affect a connected resource on a server. Assuming a transient network problem, the client tries later to repeat the operation on the server's resource. This is done by one of the *efs_agents* running on the client machine. This is the only time on the client that an agent process does client side work. If a retry attempt is not successful, either a totally dead server is declared or another retry is scheduled.

5. Summary

The choice of a stateful design approach to EFS was driven by the goal of strict preservation of UNIX filesystem semantics for access to remote files. One of the most challenging and time consuming tasks we faced during the design and implementation was the development of mechanisms to detect and recover from host and communication failures, and we have succeeded in that effort. Mechanisms are in place to allow servers to recover from dead clients without jeopardizing local access to resources, and to allow clients react to the loss of a server in much the same way as they would to a broken local disk.

Mail Routing using Domain Names: An Informal Tour

Craig Partridge

CSNET Coordination and Information Center
BBN Laboratories Inc.†

Abstract

Four of the major mail networks, the DARPA Internet, UUCP, CSNET and BITNET, tentatively plan to adopt a common naming scheme, based on one adopted by the Internet. Since the naming scheme is new to all the networks, it has required considerable conversion work, particularly in the area of mail routing. A description of the conversion process and some of the more interesting routing problems is presented.

1. Introduction

As some members of the UNIX¹ community are already aware, there have recently been a number of developments in host naming practices in four of the largest mail networks (the DARPA Internet, UUCP, CSNET and BITNET). The most significant change is that the system of appending a network appellation, such as *.uucp* or *.csnet*, to a host name to indicate the network on which the host resides is being replaced. All the networks are committed (in varying degrees of certainty) to using a new uniform naming scheme that will use the domain naming scheme recently adopted as the standard for the Internet.

The creation of a uniform name space is very convenient for mail users, but raises new problems for mail system managers. The major problems involve routing. In the past, presented with a message addressed to a user on *loki.arpa*, a mailer² needed only to look at the *.arpa* suffix to know that the destination host was on the Internet. From this point, routing was straightforward: if the mailer was on the Internet, it could deliver the message directly; if the mailer was on another network, it would send the message to a mail gateway on the Internet for final delivery.

Now, however, the mailer is presented with a message addressed to *loki.bbn.com*. The new name intentionally conveys no network information, since underlying the new naming scheme is the philosophy that users should not need to know which network a

† The author's mailing address is: c/o BBN Laboratories Inc., 10 Moulton St, Cambridge, Massachusetts, 02238. He can also be reached at the following electronic mail addresses: *craig@sh.cs.net*, *craig@loki.bbn.com*, or for the bangist world *..!{harvard,seismo,decvax,ritcv}!bbnccv!craig*.

¹ UNIX is a trademark of AT&T Bell Laboratories.

² Throughout this paper the term "mailer" is used to describe the collection of one or more programs that make up the mail delivery system on a host, or the MTA for those familiar with the X.400 terminology.

given host is on to address mail to that host. That is all well and good for users, but what's a poor mailer to do?

The answer, it turns out, depends upon the network on which the mailer resides. UUCP, BITNET, CSNET, and the Internet will probably all use different algorithms for deciding how to route messages — in large part because their network topologies are different. In the rest of this paper, I discuss how each of the networks plans to handle the problem of mail routing with domain names, insofar as the mechanisms are known, and then discuss some of the more interesting routing problems encountered so far.

2. The Networks

2.1. Internet

The Internet is the name usually given to those hosts which are connected, either directly or indirectly, via other networks to the ARPAnet and MILnet. Access to and use of the network is controlled by the Defense Communications Agency and is usually limited to organizations doing DOD-sponsored work.

The Internet is unique among the four networks in that it does not rely primarily on store-and-forward mail systems for transmitting messages. Because the Internet is a system of inter-connected local-area and wide-area networks using the TCP/IP protocols, every host can connect directly to any other host on the Internet. As a result, until recently, the method for delivering mail on the Internet was to open a connection to the destination host and pass the message on using the Simple Mail Transfer Protocol (SMTP).³ Any more complicated routing of a message was usually done by manipulating the address in the message. For example, if you wanted to send a message to *craig@loki.arpa* via *brl.arpa*, you would send the message to *craig%loki.arpa@brl.arpa*. For hosts off the network, for example *friend@oxbridge.csnet*, mailers were usually specially configured to rewrite addresses to route via mail gateways. The mail gateway for all of CSNET has been *csnet-relay.arpa* (now *relay.cs.net*), so a mailer would have rewritten the address as *friend%oxbridge.csnet@csnet-relay.arpa*. The CSNET relay “knows” how to reach *oxbridge.csnet* via PhoneNet. Mailing to a UUCP site was handled similarly, by delivering the message to one of the UUCP gateways, which would then use the *pathalias* program to determine a route (for example, a message to *user@site.uucp*, might be converted at the UUCP gateway to *host1!host2!site!user*).

However, when the Internet decided to convert to domain names the rules for delivering mail changed. To explain why this is so requires a bit of background.

For some years a plain text database listing information about every host on the Internet (addresses, services supported, host name and aliases, etc.) has been kept on a host at the Internet Network Information Center (the NIC). Every host on the network would periodically retrieve a copy of this table to learn the names and locations of the other Internet hosts. Recently it has been felt that the burden of maintaining and redistributing this database at a central point has become unreasonably high, and that the host information should instead be stored in a distributed database using domain names. Such a database has now been implemented.

³ Described in Postel [9].

The distributed database stores all of its information in entries known as *resource records* (RRs). Each RR matches a domain name with a particular piece of information about that name. For example, for *sh.cs.net*, there is a separate RR for each of its Internet addresses, its hardware, and operating system type. To get the desired information about a host, a query to the database is issued for the domain name and the type of RR of interest (e.g. RRs which list addresses for *sh.cs.net*).

The query is processed by a *resolver*, which does a “walk” through the distributed database using the domain name as a guide. The database is a tree, rooted by a server at the NIC. To resolve *sh.cs.net*, the resolver first queries the server at the NIC for information on *sh.cs.net*. The root server resolves as much of the name as it can and then tells the resolver where to go for additional information (or returns an error if based on its own information, it knows the name is bad). In the example *sh.cs.net*, the server would reply that the name appears valid but that to get a full answer, the resolver must consult the server for the subdomain *cs.net*, which is at the CSNET CIC. The resolver then proceeds to query the *cs.net* server, which has *sh.cs.net* as a leaf. The server returns the addresses to the resolver.⁴

One of the things that the domain database stores is resource records for mail transmitters (known as MX RRs). An MX RR associates a domain name with a list of hosts which will relay or accept mail for it. A domain name may have more than one MX RR associated with it. MX RRs for a given name are ordered. MXs earlier in the list are preferred over later ones. Mailers are required to try to deliver a message to each MX in succession until a successful delivery has been made, the message has been authoritatively rejected, or all of the MXs have been tried.⁵ MXs are required to pass messages on to their final destination, either through another MX or directly.

One may wonder why a new, and more complex, routing system was devised to replace the existing system. One reason is simply the desire to move routing information out of mail headers. The complexity of mail addresses in messages traversing the Internet is astonishing, and is widely perceived as unnecessary.⁶ Since much of the complexity comes from the need to do routing in message headers, moving much of the routing information out of the headers is a useful step. Another reason for the complexity of the new system is that less flexible routing systems actually turned out to be more difficult to implement.⁷ Finally, this system offers several advantages over the old:

⁴ The system may sound cumbersome to some people. In fact it often works faster than doing a linear search of the plain text database. In addition, there is a local caching mechanism that ensures that frequently used names will usually be known to the local resolver. The entire system is defined and described by Mockapetris [4], [5] and [6].

⁵ Actually, mailers need only try a reasonable subset of the MXs, where a reasonable subset is felt to be much larger than one MX. The precise interpretation of MX RRs is described by Partridge [8].

⁶ This is not a criticism of the standard for the format of Internet mail headers (see Crocker [2]), but rather a commentary on the willingness of mail systems to make unnecessary use of the rich range of formats available.

⁷ Indeed, the original plan for the domain system called for simply classifying mail information into two RR types, one for mail forwarders, which passed the messages on to the final destination, and one type for destinations, which were the host to which final delivery was to be made. This system proved to have at least two major problems. First, it wasn't quite flexible enough. Several sites wanted to have more than one mail forwarder, and had preferences about which ones to try first; the simple two type mechanism didn't support this. Second, dividing the world into two RR

- extremely flexible routing. Sites which have intermittent links to the Internet (such as networks connected by satellite or low speed phone links) can set up a system of MXs which increase the chance that, when the link is up, queued mail will get passed through.
- dynamic re-routing of in-transit mail. Mailers are encouraged to do routing at delivery time (i.e. not when the message is originally submitted or received). As a result, if a host goes down with a severe hardware failure, a change to the database can cause in-transit mail to be automatically re-routed to a backup host.
- mail relays are transparent. If *oxbridge.edu* is actually on CSNET, an MX RR that forwards mail for *oxbridge.edu* to the CSNET relay will do so "quietly". (No more *user%oxbridge.csnet@csnet-relay.arpa*.)

Currently, the Internet is in the midst of the transition to the domain name scheme. Almost all Internet mailers can interpret domain names, but only a few know how to interpret MX RRs. (The rest simply use the old system of mapping name to destination network address but use the new domain names to get the address).

2.2. CSNET

The Computer Science Network (CSNET) was established in 1981 to provide network services, primarily electronic mail access to the Internet, to institutions engaged in computer science research. Recently its charter has been expanded to include any organization doing computer-related research in the sciences and engineering (physics, chemistry, etc.). It is managed by the University Corporation for Atmospheric Research under contract to the National Science Foundation and is administered by the CSNET Coordination and Information Center (CIC) at BBN Laboratories.

CSNET member sites fall into one of two general categories: Internet sites and PhoneNet sites. Internet sites are connected to the Internet either through local links, or through use of CSNET-provided software which allows them to reach the Internet via commercial X.25 networks. Sites already connected to the Internet often choose to join CSNET because the rules for use of the Internet by CSNET members are more liberal than those for DOD-sponsored Internet users. For the purposes of mail routing, these sites behave like regular Internet sites.

However, the vast majority of CSNET members (130 of 174 sites) get mail-only service through PhoneNet. PhoneNet is a store-and-forward star network, with the central node at *relay.cs.net* (formerly known as *csnet-relay.arpa*). Sites either call, or are called, on a regular basis by the relay to send and receive messages. Data is transmitted over 1200 and 2400 baud lines using protocols developed at the University of Delaware. All messages for PhoneNet sites must go through the relay for delivery.

Because CSNET is tied so closely both technically and culturally with the Internet, it is firmly committed to converting to domain names. Internet sites will have to use domains as a requirement for future Internet use, and the CSNET CIC is strongly encouraging PhoneNet sites to convert to domain names to maintain compatibility with

types turns out to cause partial information problems (a mailer could accidentally get only the destination or the list of forwarders, and make a bad, and potentially fatal, routing decision).

the Internet.⁸

The key to making domains work for PhoneNet is making them work at the center of the star network, on *relay.cs.net*. The situation is made a bit more complex by the fact that the relay is also connected to the Internet, and must therefore coexist in both networks. Indeed, converting *relay.cs.net* to handle domains led to the discovery of some of the domain routing problems mentioned below. Happily, most of the domain conversion has been accomplished. The CSNET relay knows how to handle domain names with MX RR's on the Internet, recognizes domain names corresponding to PhoneNet sites the relay serves, and is capable of doing some limited rewriting of domain names to make mail easier to accept for sites having trouble converting their local software to use domain names. As soon as all CSNET sites have registered domain names, CSNET will have converted to domain names.

2.3. UUCP

The UUCP-based network is the largest of the three store-and-forward networks (containing approximately 6,000 hosts) and is probably best described as a loosely federated collection of hosts. Hosts usually communicate via regular phone calls, during which messages are transferred using the UUCP protocols. Links that use X.25, TCP, and other transport protocols also exist. Connectivity is variable, with some hosts communicating via dedicated lines, and others working through a single intermittent phone link to one other host. A certain amount of network coordination work is performed by the UUCP Project.

Mail routing in the network has traditionally been explicit in the address. To reach a given host on the network, a user needed to explicitly list the route from the current host to the destination (for example, *ihnp4!wjh12!harvard!bbncca!craig*). Recently the task of routing mail has become simpler with the advent of the *pathalias* program, which can take a name and, using a database, find a route through the network to the named host.

Because of the nature of the UUCP network, conversion to domain names will be voluntary. Sites which do not wish to participate need not do so. It is expected that most sites will convert to using domain names because it is convenient.

The need to retain compatibility with hosts not supporting domains has created interesting problems for authors of new UUCP mail systems. Very recently software has appeared, most notably *smail* from the UUCP Project, which can support both types of hosts. The key to the software is that while the texts of messages will use addresses of the form *user@domain*, the actual routing by mailers will continue to use the old notation based on exclamation points.⁹ Domain names will be resolved incrementally, with each host forwarding messages to another host which understands more of the domain name until the message reaches its destination.

⁸ The NIC has kindly agreed to register names for non-Internet sites such as CSNET PhoneNet and UUCP sites. This avoids the headache of different (or worse, colliding) domain names in different networks.

⁹ This system is described by Horton [3].

2.4. BITNET

BITNET, a store-and-forward network serving the academic community, communicates using RSCS, an IBM protocol, over dedicated 9600 baud lines. Each BITNET site is required to have connections to two other sites, thus providing the necessary connectivity and reliability. Sites relay messages to their final destination. Network support is provided by EDUCOM and the City University of New York.

Currently,¹⁰ BITNET still has not decided whether to use Internet domain names. A task force met in March of 1986 and, among other things, recommended that BITNET adopt the use of domain names, but this recommendation has not yet been officially adopted by the BITNET Executive Committee.

3. Interesting Problems

Readers will probably not be surprised that there are some problems posed by converting to the use of domain names. The more difficult problems observed so far do not occur when routing mail within a given network but when mail must cross network boundaries (e.g., mail sent from the Internet to a host on UUCP). Since the four networks are interconnected (which is why a consistent naming scheme was appealing in the first place), and a large amount of mail flows between them, the problems have some very practical consequences. Here are some of the more interesting problems encountered:

3.1. Naming Problems

The domain naming scheme assumes that a host's name has some correspondence with the organization operating the host. Or to put this another way, all the hosts within an organization are normally expected to have names in the same domain. However, currently it is not uncommon, particularly in university settings, for different divisions of an organization to be members of different networks and have no connections with each other. For example, almost every university computer science department in the United States is a member of CSNET. At many universities there is also a general purpose campus computing center which is often connected to BITNET. In some cases, there is no network link between the computer science department and the campus center. In the past, mail for the two sites could be distinguished by the *.csnet* and *.bitnet* suffixes.¹¹ If, for example, both are part of "Oxbridge University", and now in *oxbridge.edu*, how do we make sure that mail sent to some host in the Oxbridge University campus center doesn't accidentally get sent to the computer science center and die there?

One obvious option is to simply keep lists of exceptional cases. Where a site is served by more than one network, each network would keep track of which networks served the various hosts at the site. The level of inter-network coordination, however, not to mention the probable size of such lists, probably makes this solution unwieldy.

¹⁰ April 25th, 1986.

¹¹ Actually people tend to distinguish both by network suffix and also in the name. Sites which used the same name with different suffixes on two networks have already encountered occasional problems because mailers get confused or ignore the suffixes, causing mail to be mis-delivered. Clearly domain naming makes the potential confusion even greater.

One might also note that on the Internet, where every host routes messages itself and there are no internal relays or backbone sites, it is likely that every host on the network would have to store these lists, which is clearly unworkable.

A better option is to extend the domain names to show internal organizational divisions. Domain names can actually contain up to 63 labels (where a label is a sequence of characters delimited by dots, such as *sh* in *sh.cs.net*) and have a total length of 255 characters. Thus, using the example of Oxbridge University, a host in the computer science department could have a name ending in *cs.oxbridge.edu*, and the campus computing center machines could have names ending in *ccc.oxbridge.edu*. The networks would only need to know that mail to *<host>.cs.oxbridge.edu* should be sent via CSNET, and that mail to *<host>.ccc.oxbridge.edu* should be sent to BITNET.¹²

The divisional naming trick works well for most situations, but does have a few problems. One problem is philosophical. A stated purpose for using domains has been to spare users from having to remember on which network a given host within Oxbridge University resides. Divisional naming is analogous to inserting *csnet* (in place of *cs*) or *bitnet* (in place of *ccc*) into the domain name, thereby bending the spirit of the naming scheme. We may clothe our nakedness in the justification that divisional naming is more mnemonic or conveys additional information about the host, but in our hearts we probably know better.

The other problem with divisional naming is practical. It is quite possible to have two hosts within a division which are unable to communicate. A good example is an IBM machine, which will probably happily communicate with BITNET but may have little concept of UUCP, PhoneNet or TCP/IP. If it shares a computer room with a UNIX machine on the Internet, there is not much that a network-independent naming scheme can do to distinguish between the two hosts, except to list routes for them individually.

Obviously the best solution to this problem is for all hosts at a site to be somehow interconnected, but we are a long way from implementing this solution for all environments.

3.2. Gateway Problems

A tougher class of problems exist for hosts which reside on more than one network. The problems exist for any host on more than one network but are most acute at mail gateways, hosts which relay mail from one network to another (for example, *relay.cs.net* between CSNET and the Internet, and *wiscvm.wisc.edu*, the BITNET-to-Internet gateway).

Gateway problems appear when choosing between various routing options provided by the different networks. There are a variety of different possible scenarios; here are some of the more interesting ones discovered so far:

Imagine a message on *relay.cs.net* addressed to a host in *att.com*. Hosts in *att.com* can be reached via both UUCP and CSNET. Suppose, for a moment, that AT&T has decided that it would generally like mail from the Internet routed via UUCP instead of CSNET, and thus has listed a UUCP gateway as its preferred route in the Internet

¹² Somewhere within BITNET and CSNET, information has to be stored about how exactly to find the given host, but this information presumably exists already.

database.¹³ Because *relay.cs.net* is on the Internet, it is capable of querying the domain database and learning that the preferred Internet routing for *att.com* is via a UUCP gateway. The question is, should *relay.cs.net* make that query? Does CSNET have an obligation to consult another network about how to route messages that it can deliver internally?

One might instinctively answer that, given that we know AT&T's preferences, *relay.cs.net* should heed the expressed desires of AT&T and send the message via the Internet to the UUCP gateway instead of delivering the message within CSNET. But there is good reason not to consult the other networks.

What if AT&T wants to use CSNET as a backup for UUCP service? After all, they have both services, why not use them? Then AT&T presumably has listed *relay.cs.net* as the second choice mail relay for hosts in *att.com* (after the UUCP gateway). It is now likely that our hypothetical message on *relay.cs.net* was sent to the relay because some Internet mailer couldn't reach the UUCP gateway and has successfully used the second choice. In this situation, if *relay.cs.net* consults the Internet for routing information it will undo AT&T's attempt to use CSNET as a backup.

Of course the message for *att.com* may have originated within CSNET, in which case we arguably still want to consult the Internet for routing information. Many mailers have at least a vague notion of how they received a message, so conceivably a routing mechanism which took into account the network from which the message came could be devised: if the message is from CSNET, try routing via the Internet first; otherwise try routing via CSNET first. However, recall that AT&T still wants the message sent via CSNET if we can't reach the UUCP gateway. Therefore what the mailer must do (if the message came from CSNET), is get the Internet routing information, confirm that it can reach the UUCP gateway (e.g. try to deliver the message) and if that fails, try via CSNET. Unfortunately this suggests that the relay's mailer must do both routing and delivery at once, while the general trend has been to separate these two functions into separate modules to reduce mailer complexity.

The reader should also note that the preceding strawman examples were carefully chosen not to raise issues about how information from the different routing systems might be compared. Because the Internet database stores a ranked list of MX RRs, it can tell us that AT&T prefers to have its mail routed via UUCP. If a gateway between CSNET and UUCP existed, the routing decisions would become much trickier because the routing systems for the two networks are not compatible. On the CSNET side, the mailer knows that it can reach any host by sending the message on to *relay.cs.net*. On the UUCP side, *pathalias* can tell the mailer how many intermediate host hops are required to reach the destination. If the number of UUCP hops is greater than 1 (i.e. the mailer doesn't talk to the destination directly), there is no easy way to figure out which route is faster.¹⁴

¹³ It should be emphasized that the example routes are all purely hypothetical, and are not intended to reflect on the quality of the mail delivery systems on the different networks.

¹⁴ Even if the destination host is only one hop away, it may be faster to send via CSNET if the UUCP link is down.

It appears that the best solutions to the problems of mail gateways (without requiring large amounts of research work) are the simple ones. None of the three mechanisms described below is perfect. Indeed, none of them even attempts to determine the "best" route. They are designed to be relatively simple, easy to understand, and to have a low chance of causing mail loops. Presumably, a mail gateway can choose one of these algorithms to obtain reasonable routing performance for its particular configuration.

One solution is to establish the rule that a gateway should refuse to move mail between networks unless it must. This method works quite well if there is only one gateway between two networks. All mail destined to cross networks will be directed to the gateway. When trying to route the message, the gateway will discover that on one network it is supposed to mail to itself, while on the other network it can send the message on to the destination. The decision process is simple, requiring only that a mailer realize that a route to itself implies it should try another network. Unfortunately, the system breaks down if there is more than one gateway available to reach a given destination. There is some chance that two gateways will end up in a loop, sending mail for some destinations between each other indefinitely.¹⁵ Another problem is that hosts on one network have only a single way to get to a host on another network. If mail to *att.com* from the Internet reaches *relay.cs.net*, the relay will pass the message on to a UUCP gateway instead of delivering via CSNET, because the message does not appear to need to leave the Internet to be delivered.

A second solution is for a gateway to have a preference about which network to use. This reduces the chance of mail looping between gateways (this can only happen between gateways within the network that the gateway prefers to use, not on both networks), and the choice of networks to a destination can be exercised. If *relay.cs.net* prefers to send via CSNET, as it presumably would; mail to *att.com* that reached the relay would be sent within CSNET. The flip side of this solution is that mail will tend to stay within the gateway's preferred network. Explicit routing, such as *@uucp-gateway:user@domain*, would be required to force the mail deliverable within the preferred network to cross the network boundary.

Finally, one can use an entirely different solution, and cross network boundaries whenever possible. If a message comes in on one network, the gateway tries to send it out on the other. This strategy actually seems to have the least chance for causing mail to loop. Unfortunately, the scheme runs into other problems. It is the strategy least likely to honor network access restrictions, such as those for the Internet, because it encourages messages to go through other networks. It also does not work as well as the "choose a network" strategy for star networks such as CSNET's PhoneNet, where mail at the central relay that can be delivered within the network is always only one hop from its destination, and thus is probably best delivered internally.

¹⁵ One can, for example, imagine a UUCP gateway and *relay.cs.net* shuttling a message for *att.com* between each other because both resist letting the message out of the Internet. This scenario is, in fact, impossible because the Internet standards forbid MX hosts from sending to MX hosts less preferred than themselves. So the UUCP gateway would be forbidden to send to *relay.cs.net*. But the problem may be encountered on other networks with different routing mechanisms.

3.3. X.400

Now that we may envision several interconnected networks using the Internet domain naming standard, how well do we fit in with the the rest of the mail world? In particular, how compatible are these networks likely to be with the emerging X.400 standard? The answer seems to be that while the standards are not compatible, messages can be translated from one format to the other, and that mail gateways to X.400 networks will begin to appear in the foreseeable future.

In addition, an Internet standard will shortly be issued (if it has not been released already) describing how to convert from messages in Internet standard RFC-822 format to X.400 format and vice-versa.¹⁶ Since CSNET and most of the UUCP community also use 822 messages (or something very similar), we may hope that software implementing this standard could work on all of these networks, and that this software will appear soon.

The way in which BITNET will interface with X.400 is less well known (at least to this author), but their needs may be more pressing than that of the other networks. BITNET communicates with 200 European sites, and European countries are moving swiftly towards adopting X.400 as the mail standard. As a result, in the near future BITNET will be obliged to interface with X.400 systems, and thus it seems likely that some of the first X.400 gateways will appear on BITNET.

4. Conclusion

As the title of this paper implies, its purpose has been more informational than research oriented. The hope is that USENIX attendees will have a better sense of how the mail systems with which they come in contact will likely evolve over the next few years, and how these systems in turn, will deal with the larger outside world (e.g. X.400).

On the technical side I've only looked at the most basic question: Can mail be successfully routed between two points in the cooperating networks? Or to put the problem more bluntly, can using domain names be made to work? Happily the answer is yes. A tougher question is whether we can route mail optimally, so that it takes a minimum amount of time to get to its destination. It turns out that we don't know the answer to this question and may not for some time. It depends on a variety of factors, most notably how easily routing information from different networks can be compared. Stay tuned.

References

- [1] Comer, D., The Computer Science Research Network CSNET: A History and Status Report, *Communications of ACM*, October 1983, 747-753.
- [2] Crocker, D.H., *Standard for the Format of ARPA Internet Text Messages*, RFC-822, University of Delaware, August 1982.
- [3] Horton, M., *UUCP Mail Interchange Format Standard*, RFC-976, Bell Laboratories, February 1986.
- [4] Mockapetris, P., *Domain Names - Concepts and Facilities*, RFC-882, USC Information Sciences Institute, November 1983.

¹⁶ The author would like to thank Steve Kille for allowing him to read his draft of this standard.

- [5] Mockapetris, P., *Domain Names – Implementation and Specification*, RFC-883, USC Information Sciences Institute, November 1983.
- [6] Mockapetris, P., *Domain System Changes and Observations*, RFC-973, USC Information Sciences Institute, January 1986.
- [7] Nowitz, D.A., and Lesk, M.E., *A Dial-Up Network of UNIX Systems*, Bell Laboratories, August 1978.
- [8] Partridge, C., *Mail Routing and the Domain System*, RFC-974, CSNET CIC, BBN Laboratories, January 1986.
- [9] Postel, J., *Simple Mail Transfer Protocol*, RFC-821, USC Information Sciences Institute, August 1982.
- [10] Postel, J. and Reynolds, J., *Domain Requirements*, RFC-920, USC Information Sciences Institute, October 1984.

The AT&T Mail Service and Network

Dale S. DeJager

AT&T Information Systems
307 Middletown-Lincroft Rd., Lincroft, NJ. 07738

Network Address: attmail!ddejager

ABSTRACT

AT&T Mail is a comprehensive, nationwide, electronic messaging system that includes a full set of integrated Personal Computer and UNIX* Private Message Exchange (PMX) software. The service can be accessed from Personal Computers, from UNIX systems, or from ASCII terminals. Users can send and receive telex messages from telex terminals worldwide. Users can send electronic and paper messages to each other, and can send paper messages to nonusers. Paper messages can have subscribers' signatures and logos if desired and can be delivered either via overnight or same-day service using AIRBORNE, or via the US Postal Service. Users can retrieve their messages using text-to-speech translation devices in the AT&T Mail network. Within the AT&T strategy of Unified Messaging, AT&T Mail can exchange electronic mail with other AT&T messaging products and can also arrange to have their message waiting lamps light on certain AT&T PBX systems.

Though the AT&T Mail network consists of multiple nodes, each containing a number of UNIX System V processors, the network topology is invisible to all users of the service. The network can be used by any UNIX system as a message transport and delivery mechanism using UUCP. The network has local dial access points across the United States, and appears as one large UNIX system, called *attmail*, to any UNIX system that accesses the service. Users on registered UNIX machines may use their existing UNIX mailers, or may license PMX software from AT&T to make full use of all AT&T Mail features.

1. THE AT&T MAIL SERVICE

AT&T Mail provides electronic messaging services for ASCII terminals, Personal Computers, UNIX processors, and other hosts. Figure 1 is a graphical representation of the interconnection provided by AT&T Mail. The services currently available include standard electronic mail between users and UNIX systems, paper delivery capabilities with logos and signatures (same day, overnight, or USPS), full bidirectional telex interconnection, message retrieval from Touch-tone telephones, Collect-On-Delivery electronic messages, receipts, business forms, automatic and manual forwarding, and automatic response using a user supplied message database. Users have the option of storing messages indefinitely on the service and may also create address lists to simplify message creation and sending. Address lists may be either shared or private.

1.1 User Interface

The user model presented by the AT&T Mail service and software is one of an electronic office as depicted in Figure 2. Users who use ASCII terminals to access the service have a line-at-a-time, menu oriented, user interface available. The verbosity of the menus can be controlled so that expert users need not be bothered by menus. Online, context sensitive, help is always available.

* UNIX is a trademark of AT&T.

The large box titled "Electronic Office" in Figure 2 contains the conceptual model for the online user. Messages sent to a user are placed in the IN folder and remain there indefinitely until the user reads or deletes the messages. After a message has been read, the service continues to retain the message in the IN folder for an additional 24 hours before automatically deleting it. A user creates and edits new messages on the DESK. After creating a message, the user sends it to the desired recipients. A copy of each message sent is maintained by the service in the SENT folder for 24 hours. Messages deleted by the user are placed in the WASTEBASKET by the service. Since the WASTEBASKET is not emptied until the end of a session, the user can retrieve messages that are inadvertently discarded during the current session.

If a user desires to keep messages longer than 24 hours after transmission or reading, a FORMS/FILES option is available. This allows the user to create permanent folders for message storage. Figure 2 shows two such folders, DRAFT and HENRY. FORMS/FILES users can also create address lists which can be used to simplify the task of sending messages to people on a repetitive basis. Address lists can contain primary, secondary, blind, and paper recipients and can even reference other address lists. Furthermore the owner of an address list can specify if it should be private or public.

1.2 Forms

The ability to create and respond to forms is an elegant capability of AT&T Mail. Any user can fill out a form sent to them by simply using the ANSWER command of AT&T Mail. A form consists of fields separated by white space and colons. For example, to create a form that collects users addresses and hobbies the following would be entered by an AT&T Mail user:

Name:
Company:
Street:
City: State: Zip:
Please enter a description of your hobbies below.

:

Thank you.

When a user answers the form, the user will be prompted for each field. Note the capability to collect multiline fields using a colon on a line by itself. The AT&T Mail Forms Specification also describes how to specify a type, length, and other attributes for each field, though these criteria can be omitted as shown above. When the above form is filled out, it may look like this:

Name: John Q. Public
Company: AT&T Information Systems
Street: 307 Middletown-Lincroft Road
City: Lincroft State: NJ Zip: 07738
Please enter a description of your hobbies below.

I enjoy canoeing, rafting, and hiking. My favorite canoeing
trips have been to Quetico in Ontario.

Thank you.

1.3 Directory

The AT&T Mail service maintains an online directory of all subscribers to the service. This directory may contain listed and unlisted entries. Users with a listed entry can be looked up using their last name and optionally using phonetic matching; unlisted entries can only be accessed if the sender knows the receiver's AT&T Mail ID, also called a username. If even this latter access is deemed undesirable, users can be placed in a closed user group as described below.

Registered UNIX system users can have "Off Net User" entries in the AT&T Mail directory. This permits messages to be directly addressed to such users as though they were on-net users. The service will automatically forward messages addressed to such users to their UNIX mailbox.

1.4 Addressing

Users of the AT&T Mail service may address messages in a number of different ways. They may specify a human name as a recipient:

To: John Q. Public
To: J. Q. Public
To: Public

The response to any of the above inputs will be all of the possible matches for the input specified.

Users may request that a human name be looked up phonetically:

To: ?Publik

They may specify a unique username:

To: !jpublic

They can address a message to a UNIX system:

To: system!userid

They can send a telex message:

To: telex!123456 (ANSWER BACK/ATTN TO)

Or they can reference either a private or shared list:

To: ~privatelist
To: !otheruser~publiclist

In addition to the above, users can address messages to non-registered postal recipients and/or specify options such as /RECEIPT for recipients.

1.5 Extended Features

AT&T Mail makes available all of the normal mail handling capabilities that users have come to expect in an electronic mail system. This includes features such as answer all, answer author, and manual or automatic forwarding. AT&T Mail users can also send messages COD* and can send messages with a return receipt required.

Users may also retrieve their messages from any Touch-tone telephone using text-to-speech synthesis devices in the network and an AT&T Standard Touch-tone command set. All users of AT&T Mail have one network password assigned to them that is unique on the Touch-tone keypad. In addition, users may assign a second password to their account.

AT&T Mail has two automatic response mechanisms. Users may provision their account to automatically send a specific message back to the originator of any incoming message. This is useful for "I'm on vacation" type messages. The service will only send the message back to a specific originator once in any two week period. Users may also provision their account to send one or more messages from a user folder back to the message originator. In this case the Subject field of the incoming message is scanned for "Request:" and any messages in the specified folder whose Subject matches the incoming request are returned to the originator. Given the capability of the service to

* If a receiver chooses not to accept a COD message, the sender is billed.

transmit binary messages without having to first encode the message as printable ASCII, the autoresponse feature allows companies to set up a simple mechanism to distribute software as well as text messages.

1.6 Unified Messaging

One of the problems users encounter when using many public and private electronic mail systems is their inability to know when they have new mail. This problem is complicated by the fact that in some environments users may have to check multiple systems to see if they have mail. An architecture and strategy called Unified Messaging has been adopted within AT&T to address these problems. Introduced by AT&T in November of 1984, Unified Messaging includes a set of basic principles that underlie and direct this strategy. These principles are:

- Integrated Alerting and Notification
- Universal Mailbox
- Universal Retrieval
- Integrated Message Preparation
- Universal Connectivity

Users of AT&T Mail can arrange to have their AT&T Mail mailbox be their single, universal mailbox, or they can have all of their AT&T Mail messages sent to a different mailbox. Users can further specify that they wish to have their AT&T PBX message waiting lamp indicate when they have unread mail in their universal mailbox.

Internally, AT&T Mail uses a message protocol that is like X.400, but is extended in functionality and uses an ASCII, ARPANET-like header structure. This message protocol is an AT&T Standard called MHS.ASCII. MHS.ASCII is used by all of AT&T's messaging products, insuring that they can all communicate with each other and permitting the Unified Messaging architecture mentioned above. This means, for example, that the AT&T Office Telesystem product that provides integration of voice and data communications in the office environment can be interconnected with AT&T Mail.

Since AT&T is an active member of the CCITT Committees that specify X.400, MHS.ASCII was designed to easily permit an X.400 gateway to be implemented. The MHS.ASCII umbrella also includes the specification of objects which may be moved from one user to another. This specification permits the transfer of objects such as live text documents, spreadsheets, voice annotations and messages, as well as images. AT&T Mail fully supports the transfer of objects, though users of simple ASCII terminals may be unable to examine or manipulate such items.

1.7 Closed User Groups

The service supports the ability to create closed user groups which can be thought of as virtual private or semi-private networks of users with any levels of restrictions desired. Thus a private user group could be created where most of the members of the group can only send messages to each other, but some privileged members could be permitted to send and/or receive messages from the public users. Directory listings for such user groups can also be private or public. The model used is similar to the restrictions and capabilities available to CENTREX telephone users. Thus, any specific service element may be enabled or disabled for any individual account.

1.8 Billing Options

The AT&T Mail service has a rich variety of billing options available. Generally the charges for AT&T Mail are based only upon messages a user sends—there is no charge to check or read one's mail. An individual AT&T Mail account may be billed to itself, a master account, or a super-master account. Thus three levels of hierarchical billing are available. In addition to these three levels a fourth, lower level is available to registered UNIX systems that permits the system administrator to reallocate charges to the UNIX system users who incurred the charges. It is also possible to receive reports from the AT&T Mail billing system that permit cost allocation by project across accounts. This is useful, for

example, in an attorney's office where each attorney may have an individual AT&T Mail account and multiple attorneys desire to bill to the same client. The attorneys may simply indicate when they send messages with which project (client) a message is associated.

AT&T Mail is used to distribute its own bills. The bills may be distributed in either electronic or paper form. Users may arrange to use a major credit card for billing if they desire, in which case a reference bill is sent to them electronically.

2. THE AT&T MAIL ACCESS SOFTWARE FOR PERSONAL COMPUTERS

When used with many public electronic mail services Personal Computers simply run terminal emulation software, making a \$2000 PC act like a \$600 terminal. With some mail services it is possible to process one's mail when offline, but when the PC connects to the service it acts like a human who is typing very quickly. AT&T Mail was designed from the outset to use the full capabilities of Personal Computers. AT&T Mail Access I and II are designed to work on AT&T PC6300, IBM PC/XT/AT*, and other MS-DOS** compatible PCs. AT&T Mail Access III is for use on Macintosh§ Personal Computers. Whereas the transmission protocols used by many electronic mail services are SAP† and RAS‡, the PC interface to AT&T Mail utilizes the XMODEM file transfer protocol and is binary transparent. A specific message protocol is also defined and published which permits multiple messages to be sent and/or received to/from the service in an unattended session between PCs and AT&T Mail. This protocol is analogous to the X.400 P3 protocol.

There are currently three software packages available from AT&T for use with AT&T Mail. (Additional packages are also available from independent software vendors under the vendor software approval plan described below.) All PC software now available from AT&T allows users to read, and create their messages offline, to move messages to/from the service in an unattended manner and to maintain a private file cabinet for permanent message storage in folders named by the user. As an added benefit of using any of the Access software or PMX software, messages sent from PCs or UNIX machines cost less than messages created online with AT&T Mail.

2.1 AT&T Mail Access I

AT&T Mail Access I allows the user to create messages offline using a full screen what-you-see-is-what-you-get editor. It allows simple selection of recipients, logos, signatures, and recipient options using single function keys. Users can create, send, and fill out full screen and multi-screen forms. Communication with the service may be entirely unattended and initiated with a single function key. If users desire they may, however, connect to the service directly for use of the AT&T Mail directory and other service features such as changing autoforwarding or passwords.

Access I allows the users to attach and detach the contents of messages with one function key. This makes it almost trivial to send any DOS file such as a spreadsheet from one user to another.

2.2 AT&T Mail Access II

AT&T Mail Access II is designed for the "power" mail user, rather than the nontechnical Access I user. Access II contains all of the functions of Access I, but also contains an integrated address book, powerful macro capabilities, full VT-100 emulation, and the ability to have a menu of services in addition to AT&T Mail.

* IBM is a registered trademark of International Business Machines Corporation. IBM PC-XT and IBM Personal Computer AT are trademarks of International Business Machines Corporation.

** MS-DOS is a registered trademark of Microsoft Corporation.

§ Macintosh is a trademark licensed to Apple Computer, Inc.

† Send And Pray

‡ Receive And Swear

2.3 AT&T Mail Access III

AT&T Mail Access III is an electronic mail software package designed especially for the Apple Macintosh. Though Access III has all of the features of Access I, the user interface is tailored for the MAC. Thus it makes use of a mouse with pull-down menus and a highly icon oriented screen. Users can utilize all of the cut and paste functionality that the MAC clipboard and scrapbook make available.

2.4 Vendor Software Approval Program

To encourage vendors to build mail software and applications that are compatible with AT&T Mail, AT&T has instituted a vendor software approval program. For a nominal fee, anyone may receive a copy of the message interface specification which describes in detail the message protocol to use when submitting and receiving messages to/from AT&T Mail. Also specified are the forms definition language and the precise XMODEM protocol that is supported for exchange of messages with PCs. The vendor is also sent information detailing the approval criteria.

If a vendor decides to use the specification to implement a product, the vendor may submit the product for approval to AT&T with a approval fee. If AT&T determines that the product is approved, the vendor will be so informed and may advertise that fact. This process ensures that the software for use with AT&T Mail properly interfaces to AT&T Mail.

3. THE AT&T MAIL PRIVATE MESSAGE EXCHANGE SOFTWARE FOR UNIX SYSTEMS

Software now available for UNIX systems maintains the same user interface as that available on PCs using Access I. It is also possible to utilize the PC software in conjunction with private UNIX systems. There are four different software packages available for UNIX systems depending on the system configuration. All of these packages can operate on the same UNIX file cabinet, and address lists can be shared between users if desired. For any of the PMX products, if the user requests a feature that is not provided by the local processor (such as paper or telex delivery), the request is automatically passed to the AT&T Mail network service for handling. (Note that due to the capabilities of PMX and the network service, telex messages can be addressed directly to/from UNIX users' mailboxes.) The only significant user visible difference between using the Access products with the AT&T Mail network service versus using the the PMX products is that addresses entered by the user are relative to the private system rather than the public service. This is consistent with the philosophy that addresses are always relative to the system to which the user is connected.

Figure 3 graphically depicts each of the Private Message Exchange (PMX) products. Figure 4 shows how UNIX processors, Personal Computers, and the AT&T Mail service work together to provide a total end-to-end solution for users. Note that all AT&T Mail capabilities are available to any UNIX system user, but the PMX product line provides a simple and consistent user interface for electronic mail processing.

3.1 PMX/TERM

PMX/TERM is a software package that runs on UNIX processors and gives CRT users connected to the UNIX machine the same, full screen, user interface as that provided by Access I. While all of the features that Access I provide are available to the user, the user also has the ability to be notified of new incoming mail while processing other mail. In fact, new incoming mail can be automatically received with no user initiated action, if the user so desires. Note that because PMX/TERM is for ASCII terminals, the user must be online while processing mail. Since PMX/TERM uses Curses, virtually any terminal can be supported. (A reasonable mapping of the keyboard is made using the Curses 5.3 capabilities that allow for numerous function keys. For those terminals without function keys, a generic escape mechanism is available.)

3.2 PMX/PC

PMX/PC allows PCs running any of the Access products described above to be used with a private UNIX system in addition to the AT&T Mail service. The user merely needs to use a different logon script in the Access package to access a UNIX machine instead of the AT&T Mail network. PMX/PC,

once invoked, emulates the commands of the network service used by Access I, II, and III. Because of this, the PC software is unaware of the fact that it is communicating with a private machine.

3.3 PMX/LAN

PMX/LAN allows users with PCs on a 3B-Net or STARLAN local area network to have the same user interface and mail capabilities as those on independent PCs or ASCII terminals. PMX/LAN is a mail server that processes the mail for PC User Agents. The software that runs in the PC can be loaded from the server. It is similar to the Access I Software, but has been modified to take advantage of the file sharing capabilities and high speed link in the LAN environment.

3.4 PMX/MAILTALK

PMX/MAILTALK is in the suite of PMX programs and permits users to retrieve their mail from a Touch-tone telephone. Unlike the other PMX products which are available now, AT&T is evaluating the market demand for PMX/MAILTALK before making it generally available.

3.5 Binary Mail

Note that all of the PMX products use the standard UNIX system mail command as the mail delivery agent. All of the PMX products also retrieve messages from the standard /usr/mail UNIX system directory. The standard UNIX System V *mail* command, however, does not permit users to send binary messages without corrupting the receiving user's mailbox. A version of the mail command that does not have this problem, but is upward compatible with the standard mail command is made available with any of the PMX products. This mail command inserts a Content-length: header line in incoming messages and uses this line to find the start of subsequent messages when parsing a mailbox. If no Content-length: line is found, the old rules of scanning for a line that starts with "From " is followed.

3.6 Vendor Software Approval Program

Vendor supplied software for the UNIX environment may be approved using the same procedure as that described for PCs above. In this case, however, the XMODEM protocol is not used since UNIX systems communicate with AT&T Mail using UUCP.

4. THE AT&T MAIL NETWORK

Figure 5 depicts the AT&T Mail network which currently consists of a number of processors distributed between three node locations nationwide. At each node there is a DATAKIT Virtual Circuit Switch (VCS) which provides wide area networking capabilities for the mail network. The DATAKITs are connected together using 56 kilobit DDS circuits and are also connected to an internal AT&T DATAKIT network. All incoming access and interprocessor communications is via this DATAKIT network.

Users and UNIX systems can dial any of over 40 different local access numbers, or a nationwide 800 number. Regardless of which number is dialed, the users are unaware of the network topology, or the specific machine that is serving their mailbox. UNIX systems see the network as one UNIX system with the node name of *attmail*. Nodes and processors can be added to the network as load demands without affecting any user addressing or interface. The network is designed to permit the failure of any single network component or processor without loss of service to the users. In the event of a failure, the worst that the user will typically see is a disconnected session, requiring the user to dial in again.

The flexibility described above is provided using many of the capabilities of DATAKIT. In particular, every physical processor in the network identifies itself to the DATAKIT to which it is connected by providing a list of virtual machine names that it is serving. When a specific physical machine in a node fails or is taken down, the remaining processors each take over part of the load that was being processed by down machine. This is accomplished by having each operating machine provide the DATAKIT with the virtual machine names that it is now serving in addition to its standard list. Typically a virtual machine corresponds to a service or a portion of a service. (AT&T Mail is not the only service that can be provided on the network). For the AT&T Mail service, there are a number of

virtual machines in the network. Each virtual machine provides service for a large number of users. Each end user or UNIX system is assigned to one virtual machine somewhere in the network.

There is one special virtual machine on each processor that provides normal UNIX service for administration. (Registered users of AT&T Mail have no access to the underlying UNIX systems.) There is also one, special, virtual machine in each node that provides a key service called "authentication". All incoming calls to the AT&T Mail network are automatically routed to this "auth" server which prompts the user to log in. Once the user's username and password have been validated, a network wide database is consulted by the auth server to determine the services that are available to that specific user. If more than one service is available the user is prompted to select one. A "call" is then placed via the DATAKIT from auth to the virtual machine providing service for the user. Auth informs the destination virtual machine of the incoming call and then "splices" the calls together, dropping out of the circuit until call termination. When the user is finished with a given service, the call is rerouted to auth and, if a list of services are available, the list is again presented to the user.

Note that the network architecture described above permits users to login to the network once and switch between services without logging-in to each service. The network database mentioned above is maintained centrally and updated nightly without affecting online uses.

AT&T Mail currently has one large print site with multiple, high speed, laser printers. Though the service design permits multiple print sites, with routing based on the zip code of the recipient, AT&T Mail is able to achieve rapid delivery for all forms of paper mail using AIRBORNE to deliver overnight mail directly and to deliver sacks of US Mail to regional post offices. For same-day delivery a network of AT&T group 3 FAX machines are utilized at strategic AIRBORNE locations.

5. OPERATING THE SERVICE AND NETWORK

The AT&T Mail service and network is remotely operated from one central site in New Jersey. From this location all network reconfiguration, provisioning, and software maintenance is performed. New software releases are distributed to the network processors electronically. Account management and customer registration is a quick and highly streamlined process resulting in the automatic generation of network directory changes and welcome kit cover letters. The updates to the network directory are automatically distributed on a nightly basis to all of the nodes. Similarly all of the billing data is automatically collected for processing. The only items that require on-site personnel are operations such as tape mounting/unmounting and physical network changes.

6. AT&T MAIL DIRECTIONS

AT&T is committed to international message standards and access protocols. Therefore AT&T Mail will provide X.25/X.29 national and international access as well as an AT&T X.400 gateway. Also under development is the ability for AT&T Mail to accept troff, ditroff, and postscript files for printing on AT&T Mail's laser printers. Additional host interfaces, such as for IBM systems, are already in trial with selected companies.

7. CONCLUSION

AT&T Mail was publicly announced on February 25, 1986. The service is heavily used, both internally and by many corporate customers. AT&T is unique in the industry in its ability to provide an end-to-end electronic messaging system that includes personal computers, office computers, large computers, a nationwide network and service, and software to run on the processors located on the customer's premises, providing both public and private electronic mail solutions. As illustrated in Figure 6, AT&T Mail is a crucial network product that provides application solutions and bridges the public and private electronic mail markets.

AT&T MAIL

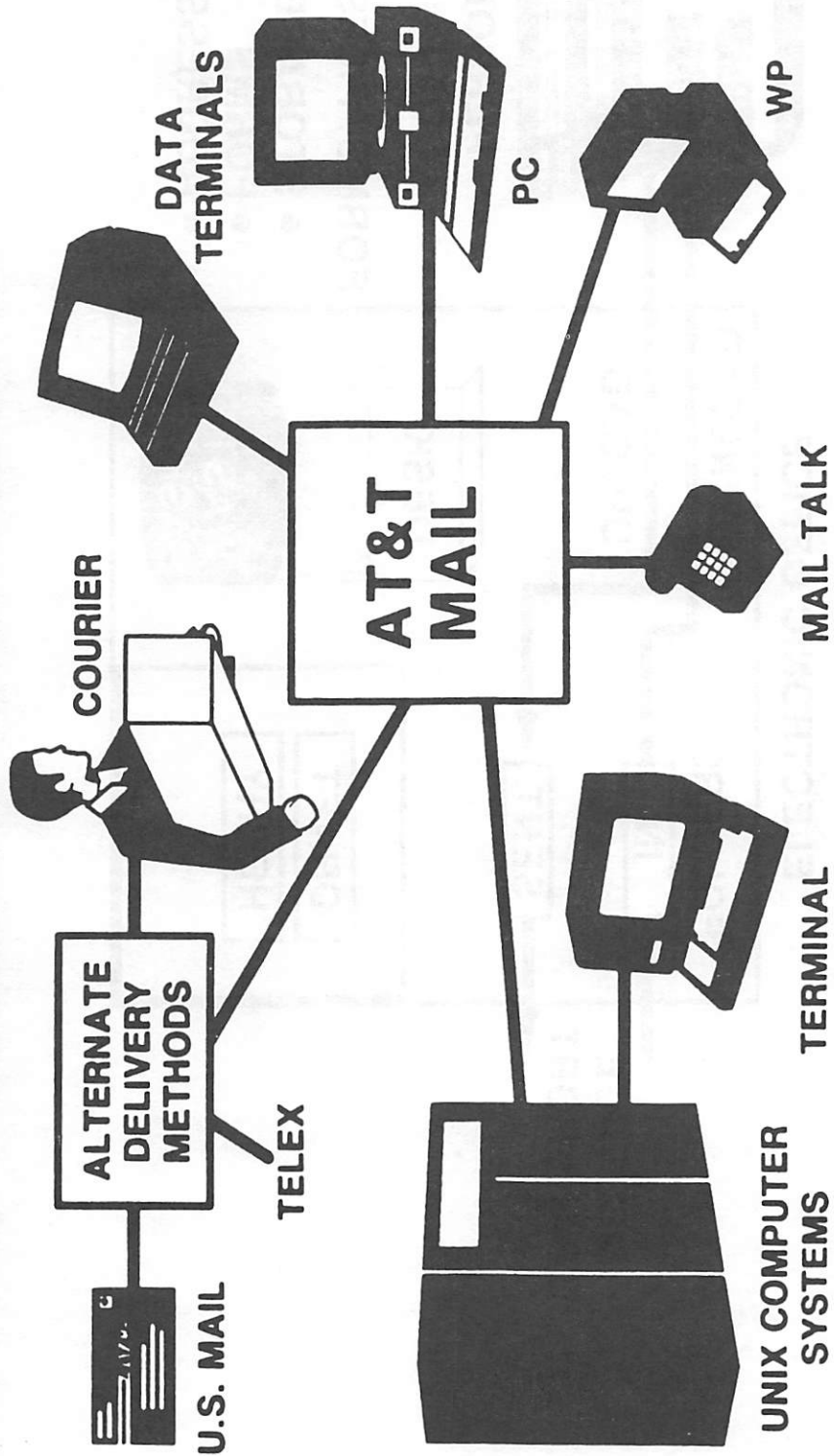


Figure 1

User Model

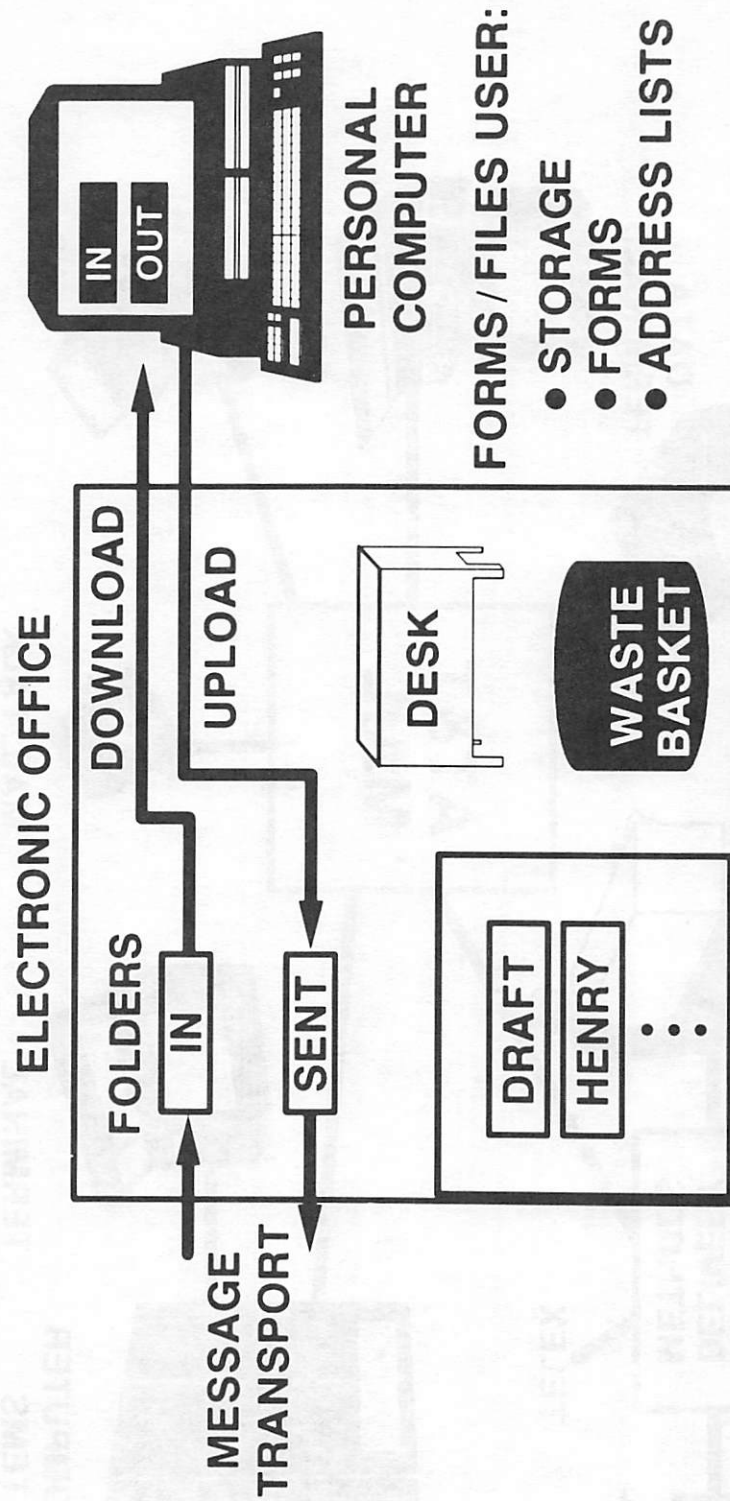


Figure 2

Private Message Exchange (PMX)

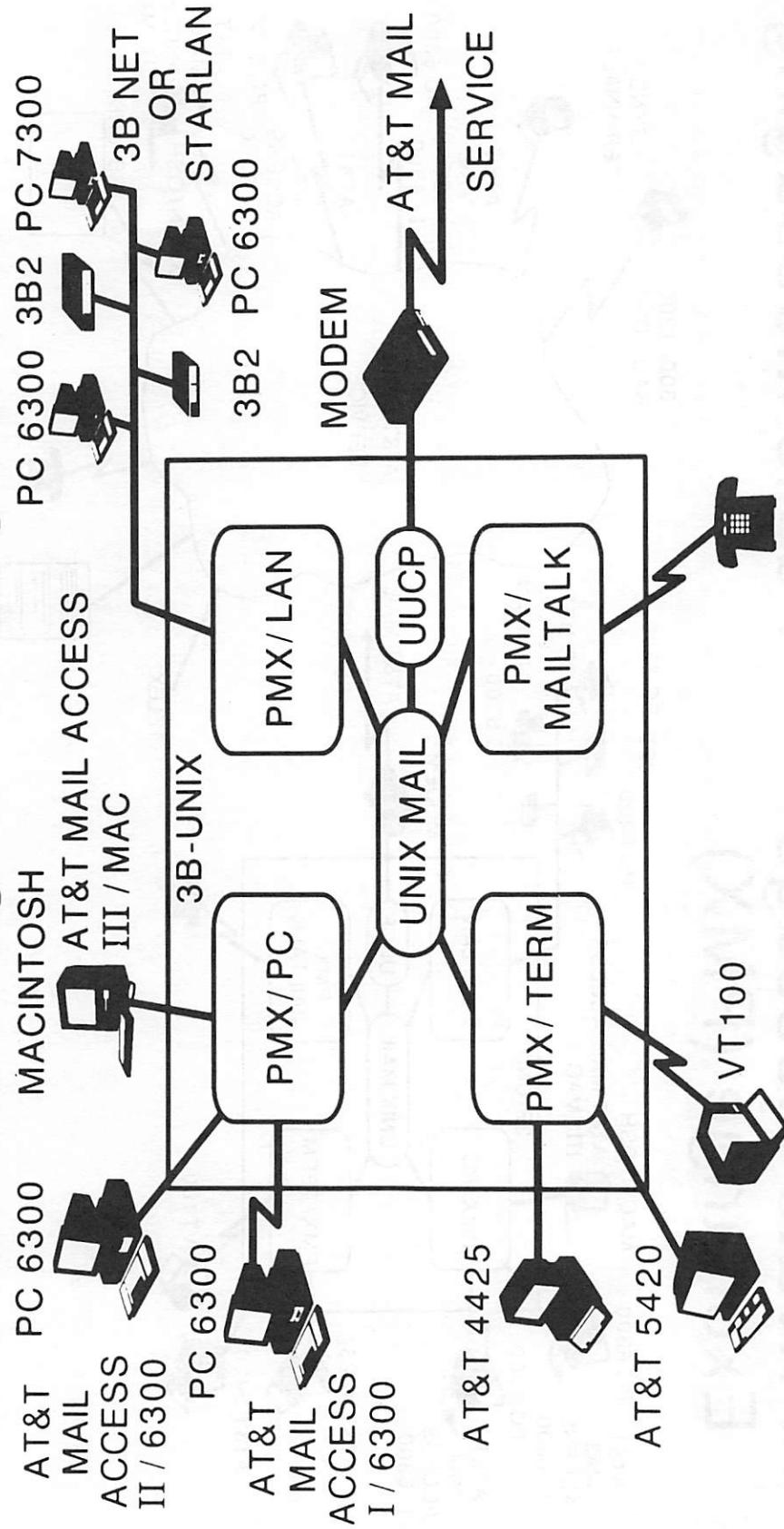


Figure 3

Private Message Exchange (PMX) Public Messaging

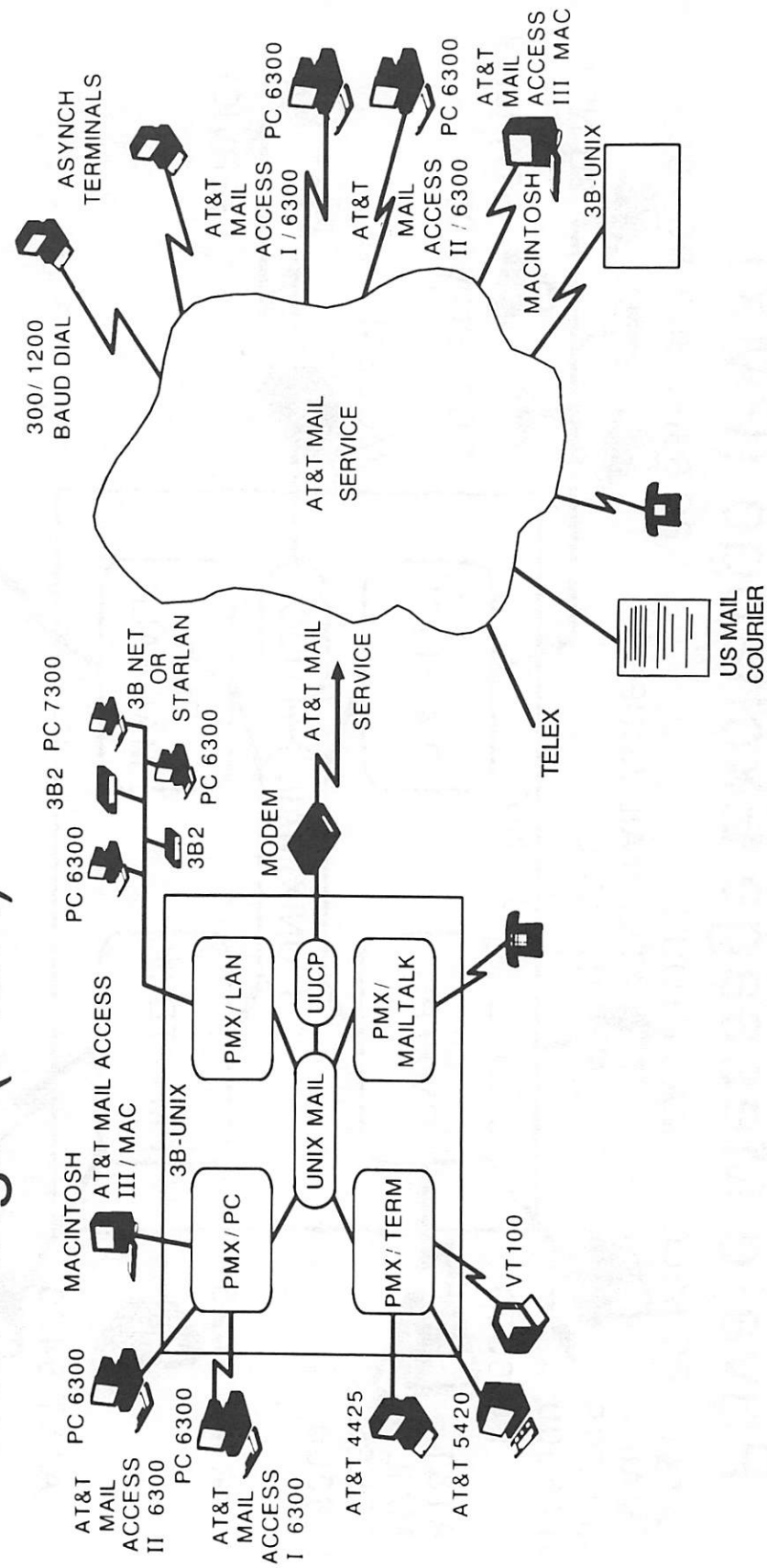
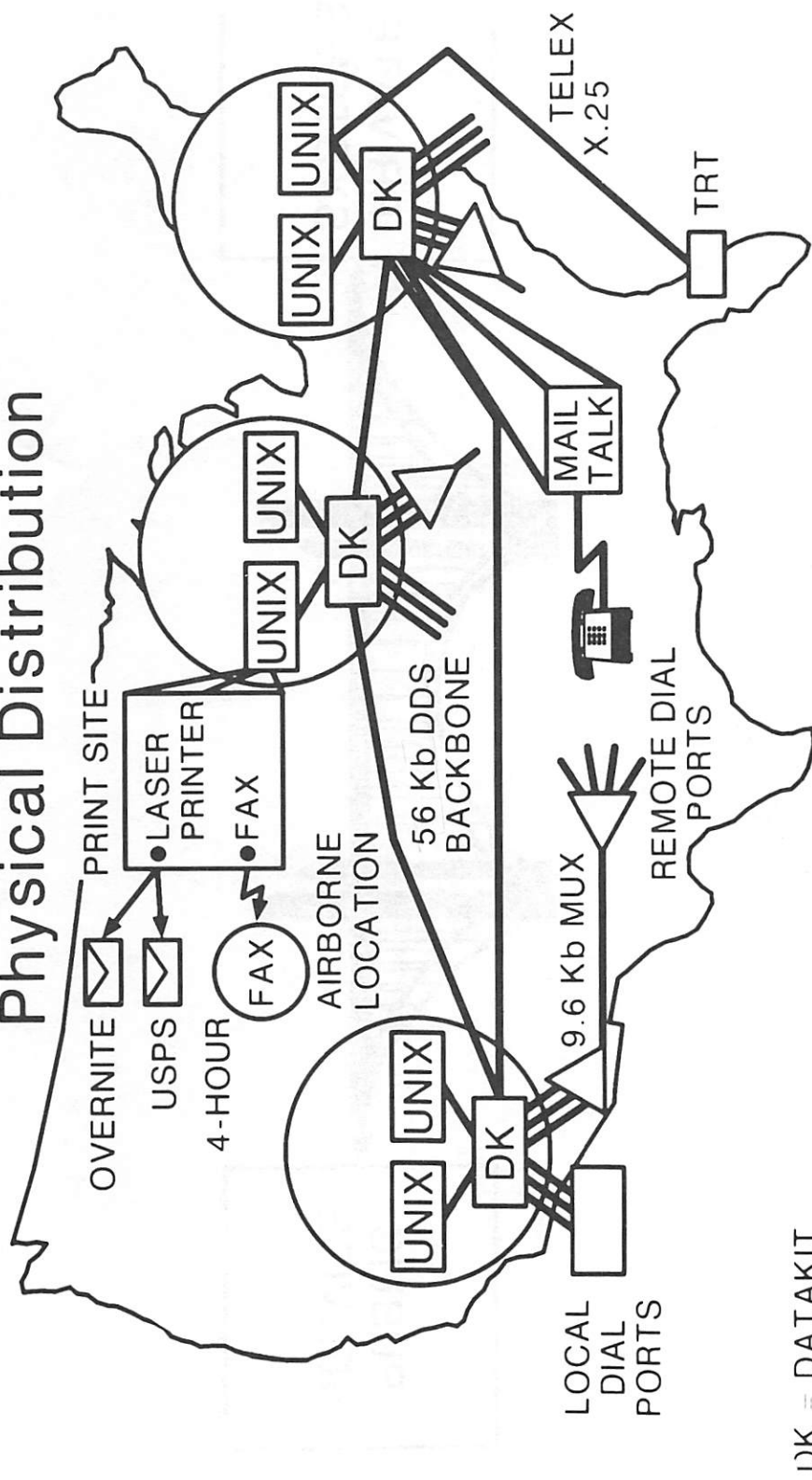


Figure 4

Network Architecture: Physical Distribution



DK = DATAKIT

Figure 5

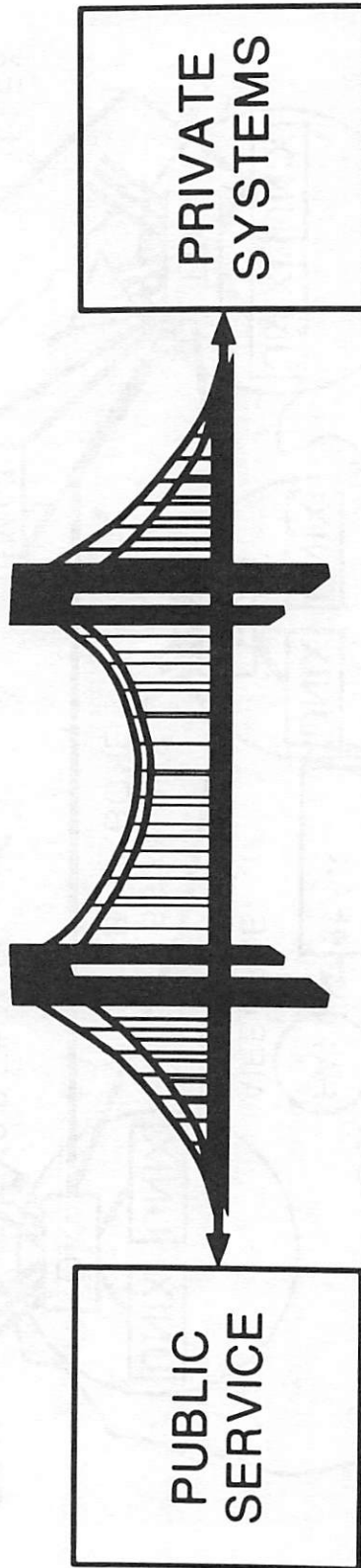


Figure 6

A Mail File System for Eighth Edition UNIX

David Hitz

`/mail/princeton/hitz`

Peter Honeyman

`/mail/princeton/honey`

Computer Science Department

Princeton University

Princeton, New Jersey 08544

INTRODUCTION

In this project we merge electronic mail addresses into the UNIX file system name space. Mail is sent by manipulating files in the **Mail File System** (MFS) of the local machine. We first describe a metaphor in which files represent mail destinations, and then examine an implementation using Eighth Edition UNIX file system types. Finally we explore extensions to the system that allow efficient delivery of messages with *multiple* destinations. By harnessing the well-behaved naming conventions of the UNIX file system, the MFS helps clarify the mail name space.

THE METAPHOR

Consider equating files in a local machine with mailboxes for users on other machines. To send mail, a process creates files and writes messages into them. When a file is closed, it is automatically removed from the file system and its contents are delivered. Thus, there are no permanent files in the MFS; files exist only while they are open. The name of the file specifies the destination. For example, with a MFS mounted on `/mail`, the following command sends mail:

```
$ cp message /mail/allegra/uw-beaver/microsoft/robertre
```

In addition, the system uses *pathalias*,¹ so

```
$ cp message /mail/microsoft/robertre
```

also works. Since MFS files are accessed with standard system calls, any program that manipulates files, from *cat* to *vi*, may be used to send mail.

The metaphor offers important advantages in clarifying electronic mail naming conventions. With file names representing mail addresses, standard file system conventions apply: paths are specified from left to right and `/` is the only separator. Of course, file names may contain `@`, so one could create `/mail/honey@down`. But this makes no more sense than specifying `/usr/bin/grep` as `/usr/grep@bin`. Nevertheless, in the current implementation, addresses with `@` or other mail syntax separators do work. A final advantage is that hiding the delivery agent in the operating system protects it somewhat.

A difficulty arises with the metaphor because some operations on regular files have no obvious

¹ P. Honeyman and S.M. Bellovin, "The Care and Feeding of Relative Addresses," in this proceedings.

analog for MFS files.² For instance, what should

```
$ ls /mail
```

produce? A list of local users and neighboring machines? If so, then what should listing `/mail/neighbor` produce? In principle, user and connectivity information for the entire network could be made available; in practice, this would be prohibitively expensive. Every `ls` would require accessing a large data base; keeping the data base up to date — especially for user information — would be administratively impossible. Also, since mail connectivity does not have a tree structure, the concepts of parent and child become muddled.

Therefore,

```
$ ls /mail
```

lists nothing. It appears to the user that he creates invisible files in invisible directories. Fortunately, directories with write but not read permission offer some precedent for this in the regular file system. A user can create files in such directories even though their contents cannot be listed.

A more serious problem with the metaphor arises when two people simultaneously send mail to the same destination. Clearly, if two people have `/mail/hitz` open at the same time, their messages should not be interleaved. The solution is to have the messages go into separate files that happen to have the same name.

Thus, in the MFS, files with different inode numbers may have identical names. At first this seems ugly, but the problem has little practical impact. Every call to `open` returns an empty file into which a user can write his message. He is unaware that another user may have opened a different file with the same name. Since files disappear as soon as they are closed, they cannot be opened a second time. Thus, non-unique file names create no ambiguity.

IMPLEMENTATION

We implemented the MFS using the Eighth Edition network file system (NETFS) as a model.³ When a MFS is mounted, a stream to a server process is associated with the mount point. The kernel passes all MFS file system requests down the stream to the server process, which handles them and then passes the results back up the stream to the kernel. Thus, this project consists of two parts: creating a new file system type in the kernel, and writing the server that actually does the work.

Kernel file system

The kernel code for the MFS is almost identical to that for NETFS. Only kernel `namei()` required modification. In NETFS, the kernel passes path name components to the server one at a time, but this cannot work with the MFS. The server has no way of knowing whether a particular component is a machine name, in which case it should be identified as a directory, or a user name, in which case it should be a file. To circumvent this problem, the MFS passes the path name down to the server all at once. The last component is the user name.

Server

Unlike NETFS which locally recreates a remote file system, the MFS is completely self-defined. A file system's main functions are to associate file names with inodes and to handle reads and

² R. Pike and P.J. Weinberger, "The Hideous Name," in *Proc. Summer USENIX Conference*, Portland, 1985.

³ P.J. Weinberger, "The Version 8 Network File System," in *Proc. Summer USENIX Conference*, Salt Lake City, 1984.

writes on those inodes, so a self-defined file system must have strategies for each. Since the metaphor calls for every *open* to return an empty file, the file name to inode strategy is simple: each *namei* call creates a new inode no matter what file name is specified.

To accommodate writes to an inode, the server stores the data in a temporary file in */usr/spool/mfs*. When a MFS file is closed, the server removes it from the MFS and mails the associated temporary file. The send function of Presotto's UPAS mailing system provides the low level interface to the network.⁴ The server could handle this interfacing itself, but we see no point in duplicating already completed work.

These strategies result in a simple system. Because each *namei* call returns a new inode, privacy can be assured. Another simplification is that files cannot be multiply opened. Performance is not an issue with this system because so little work is done.

EXTENDING THE METAPHOR

The system as described — and implemented — so far has one major weakness: there is no way to specify multiple destinations for a single message. Of course, one can copy the same message into several different MFS files, but then the mailer cannot take advantage of the fact that messages are identical. To send a message to multiple users, each on the same remote machine, it is preferable to send one copy to the remote machine, which then makes a separate copy for each user. This is also recommended for destinations on different machines if the paths share a common prefix. Many mailers shun this possible savings.

Ordinary mailers are programs, not file systems, so specifying multiple destinations is syntactically easy: each argument is a separate destination. The MFS requires a different strategy. Since *link* indicates that two different names both refer to the same object in a disk file system, *link* should also be able to indicate that two names refer to the same message in a MFS. One might send two copies of *message* with:

```
$ cp message /mail/hitz
$ ln /mail/hitz /mail/honey
```

Unfortunately, our initial definition of the MFS allows separate files to have the same name, so any technique that attempts to identify messages by name is doomed to ambiguity. In addition, the MFS removes files as soon as they are closed, so */mail/hitz* would disappear before *ln* could execute.

A second method is to access a message according to its inode number in the MFS. The system call *link("/mail/00013", "/mail/pep")* would indicate that the message with inode number 00013 should also be sent to pep. This is cumbersome because the process must first call *fstat(2)* to find the file's inode number and then build a name from that. The *fstat* call prevents use of this method from the shell.

Instead, we suggest the following: rather than sending a file's contents as soon as the file is closed, the MFS holds the message for a brief period, perhaps a minute, computing a checksum on the contents in the interim. If new messages with the same checksum arrive during that time, then the MFS recognizes that the messages are identical and sends just one copy instead of many.

For local machines, mail may be sent immediately to avoid delay. For non-local machines, the small additional delay is insignificant. This opens the door to the possibility that different messages may by coincidence have the same checksum; we view this as an opportunity for fun in electronic mail. For extreme correctness extremists, the server could *cmp* files with matching checksums.

⁴D. Presotto, "Upas — A Simpler Approach to Network Mail," in *Proc. Summer USENIX Conference*, Portland, 1985.

CONCLUSION

It is interesting that the UNIX file system can accommodate the mail name space, since the two seem — in some ways — very different. Regular file names refer to resources: data or devices. If two processes simultaneously open one resource for writing, there will either be interference between the processes or one of the opens will fail. In contrast, names in the MFS refer to destinations. Processes writing to the same destination are not at all related: neither interference nor failure of an open is justified. This is why the MFS must allow different files to have the same name.

Actually, both `/dev/tty` and the Eighth Edition `/dev/fd` files offer precedent for `/mail`. There is no interference between two process that open `/dev/fd/0`; they each read from their own standard input. Just as one process cannot *link* or *open* another's `/dev/tty`, one process cannot *link* or *open* another's MFS files. The MFS offers access to a much larger name space than `/dev/fd`, but it is similar in that the name spaces of separate processes do not overlap.

The MFS seems to be part of a new class of files that is developing in UNIX: files that have independent existence for each process that refers to them. This new class may be able to unify the access methods for a broad range of services that have not previously been incorporated into the file system.

Shared Libraries on UNIXTM System V

James Q. Arnold

AT&T

ABSTRACT

As people move the UNIX¹ System to smaller machines, it becomes increasingly important to use disk space, memory, and computer power efficiently. A shared library can offer savings in all three areas. By consolidating multiple copies of library routines, a shared library can make executable files smaller on the disk, make processes take less memory, and reduce paging or swapping activity.

This paper describes a shared library design that lets existing application source and object code use shared libraries. By extending current mechanisms—instead of inventing new ones—the shared library facility preserves the value of existing application code while offering additional benefits. A shared C library, for example, shrank 115 system programs an average of 11,000 bytes per file.

1. INTRODUCTION

Executable files on the UNIX System traditionally have held all the code and initialized data for process images. One layer of sharing exists in the UNIX System: processes created from the same executable file share one copy of the program text in memory. But, if several programs use the same library routines, each executable file holds its own copy of these routines on the disk, and each process has its own copy in memory. A *shared library* lets multiple processes—from different executable files—share a single copy of library code.

Shared libraries make programs' sizes on disk shrink, because executable files no longer hold the code for library routines. Combining numerous copies of common routines also reduces the entire system's memory requirements. Moreover, a shared library lets one fix library errors without rebuilding executable files. By installing a new shared library, old executable files automatically use the updated library.

Before going any farther, I should summarize the properties of our shared library design.

- *Application source code compatibility.* No application source code changes are needed to use shared libraries.
- *Application object file compatibility.* We did not change the C compiler's code generation either for user code or for shared library code. Thus previously existing relocatable object files can be linked with shared libraries.
- *Multiple shared libraries.* A process can use several shared libraries simultaneously.
- *No special permissions.* Anyone may build and use a shared library, without needing privileged commands.
- *Limited library source code changes.* Although one can write library code that works unchanged in either shared or unshared libraries, some minor source changes may be necessary to transform existing unshared library code into shared library code. We were willing to impose minor changes on library developers to maintain compatibility for applications.

1. UNIX is a trademark of AT&T.

- *Static symbol linking.* The link editor resolves programs' references to shared library routines; a process spends no run time finding shared library symbols.

Our implementation divides responsibilities between the *host* and the *target* machines, providing development and execution respectively. One can use the same machine for both, but development and execution occur separately and need different capabilities.

2. TARGET LIBRARIES

Processes execute on the target machine, perhaps calling shared library code along the way. The target portion of a shared library provides the run time services.

2.1 File Format

Target shared libraries use the same format as executable files, although a different magic number² distinguishes them for the operating system. To provide run time services, each target library has its own shared text and private data regions in memory. Processes normally have text and data regions themselves. Thus if a process uses three shared libraries, for example, its address space will contain four shared text regions, four private data regions, plus a private stack region.

2.2 Attaching Libraries to Processes

When the operating system executes a new program, its process loader, called *exec*, reads the program file to create the process's memory image. The operating system saves some information about the old program, discards the old memory image, builds the new memory image to match its program file, and then transfers control to the new program.

Executable files hold sections to describe the program text and data regions. We added another section to hold the file system path names of all required shared libraries. During *exec* the operating system reads the library section,³ extracts the path names, and attaches the requested shared libraries to the process. If an executable file does not have a library section, it does not use any shared libraries.

We added no new system calls; so the process itself must do *nothing* special to use shared libraries. Executable files tell what libraries they need, and the operating system makes the libraries available at run time. The kernel uses the same code to load executable files and shared libraries.

2.3 Run Time Properties

Because a target shared library uses the same file format as executable files, the operating system reuses its *exec* code to load shared libraries. Even though shared libraries are not processes themselves, some standard program and process properties naturally apply to them.

First, the operating system enforces file system permissions. Just as one must have permission to execute a program, one must have permission to use (execute) a shared library. These permission checks obey the normal rules.

Second, the kernel arranges to share multiple requests for processes' and libraries' text regions. After creating the initial memory image of shared library text, the system automatically reuses that image for future processes. The library regions exist as long as any process needs them.

2. A field in an executable file holds a *magic number*, which should be one of a few known values. Those values are unlikely to appear at the same position in non-executable files, thus giving a simple sanity check during process loading.

3. The library section resides in the file but does not reside in memory during process execution.

Third, shared libraries have private data regions, thus precluding interference among processes. Processes always have private data regions and stacks. Libraries are not separate processes, so they share their client processes' stacks at run time.

Fourth, the operating system protects the files from which processes have been executed. When a program is active, one cannot remove or write the contents of the associated executable file. Shared libraries' files are similarly protected from tampering.

Lastly, the operating system uses its standard paging strategies for shared library regions. If many processes are actively using a library, the active pages of its text region remain in memory. Conversely, the inactive pages become candidates for reuse.

2.4 Text Symbols

I mentioned that we use static linking to resolve references to shared library symbols. This means those symbols have fixed addresses, which the link editor copies into executable files. Fixed addressing causes no problems when all symbols reside in a single executable file, but shared libraries create inter-file dependencies. We therefore wanted a way to control shared library symbol addresses, obviating the need to rebuild executable files for new versions of a library.

We use a *branch table* to insulate application programs from internal shared library changes. When a function resides in a shared library, its address corresponds to a jump instruction whose destination is the function's "real" code. Each library's branch table resides at the beginning of the text region, and each function in a library occupies a fixed branch table entry. Thus the public function addresses do not change, even though private addresses do.

Branch Table		
a:	jump	a_code
b:	jump	b_code
c:	jump	c_code
Real Code		
b_code(...)	{	... }
a_code(...)	{	... }
c_code(...)	{	... }

Figure 1. Overall library structure

A library developer controls the symbols' order in a branch table. One easily can change the implementation of library functions without altering the visible symbol addresses. A utility program builds the branch table automatically, creating the jump instructions and new labels for the real functions' code. Thus a library developer writes function definitions for a, b, and c. The utility program creates the symbols a_code, b_code, and c_code.⁴

2.5 Data Symbols

Shared library symbols have fixed addresses. A branch table transparently supplies one level of indirection for text symbols, because the table's functions name executable code. When a program calls a function in the branch table, it doesn't know—or care—how the function is written. Conceptually, the branch table adds one instruction to the function's code. This indirection allows internal library code changes without affecting the external interface.

4. I've used these symbol names for illustration only. The utility program creates assembly language names that are not available in the C language, preventing name space pollution.

Unfortunately, data have no branch table equivalent. C compilers typically generate code to reference external and static data symbols directly, without implicit indirections. When an executable file references a data symbol, the machine instructions normally hold the object's address. Furthermore, when the compiler is generating code for an executable file, it cannot know where data objects will reside at run time. Some may reside in the executable file, and others may reside in shared libraries; but the final resolution is unknown until the link editor builds the executable file.

Although we could have extended the language to control code generation, we wanted strict source code and object code compatibility for application programs. Thus we decided to permit shared library data in spite of the complications. A few data structuring techniques help library developers keep data addresses from changing between versions, but data changes have more restrictions than text. Practically speaking, these data restrictions affect library developers but not application developers. Because of the potential for change, we strongly encourage library developers to limit or avoid external data symbols.

3. HOST LIBRARIES

Application programmers use host shared libraries just as they do other development libraries. For example,

```
cc program.c -o program -lX
```

builds the output `program` using library `X`. Application programmers do not need to know whether `X` is shared or unshared.

Briefly, a development library is a collection of functions and data that developers can use. A special file, called an *archive*, lets the link editor select the functions and data an executable file needs. The link editor does not copy unreferenced archive members to executable files, keeping executable files smaller than they would otherwise be.

3.1 Shared Member Contents

Regular archive members hold the text and data for the corresponding C code. Each member typically corresponds to one C source file, compiled to relocatable object code. When the link editor extracts a relocatable archive member, it copies the text and data to the output file. This works fine for a single executable file, but one of our main goals for shared libraries was to eliminate the executable files' copies of library routines.

One builds target shared libraries from relocatable object files too. After building the target shared library, all visible library symbols have fixed, known addresses. When one builds an application with a shared library, only the addresses need to be copied to the executable file. We therefore remove the files' text and data before archiving them in the host.

Although a host library's archive has members that correspond to the target's relocatable files, the archive members are "empty." We create the archive members by looking up the target's symbol addresses, creating absolute definitions in the members' symbol tables, and deleting the members' text and data. When the link editor extracts a host member, it sees absolute symbol definitions to resolve references, but the member contributes neither text nor data to the executable file being built.

3.2 Mixing Shared and Unshared Members

Because the host shared library is a standard archive and the host library members are relocatable (albeit empty) object files, one can mix shared and unshared members. We used this technique when we built a shared version of the standard C library. Although we didn't want to include all the regular C library in the target shared C library, we did want the shared and unshared libraries to be compatible. Consequently, the final host shared C library has shared members for the routines in the target and unshared members for the routines not in the target.

3.3 Building Executable Files

Executable files that need shared libraries hold a special section, as I described in §2.2. This section tells the operating system what libraries to attach for the process. Each host shared library contains a special member that defines its own information for the library section. When a program references any symbol from the host archive, the link editor loads the special member into the executable file.

If an executable file directly uses several shared libraries, it clearly will hold all their special information. Having shared libraries reference other shared libraries might seem to complicate things. We resolved the issue by forcing all library requests to reside in the executable file. For example, if a program uses library A, which in turn uses B, then the program is forced to depend on both A and B.

Two important properties arise from this choice. First, it limits the stack growth in the kernel. When the process loader looks at an executable file, it immediately knows how many libraries the file needs. It does not have to trace library chains of A needing B needing C needing Second, the behavior preserves the way we currently handle regular libraries that need other libraries. Programmers do not have to learn a new set of rules for dealing with shared libraries, and we did not have to change the link editor algorithms for resolving symbols.

4. IMPORTED SYMBOLS

Shared libraries normally are reasonably self-contained. As an absolute executable file, a target shared library can have no unresolved references. Nonetheless, a shared library can *import* symbols, thus giving it a way to reference functions and data outside its immediate control. Import capabilities markedly improve shared libraries' versatility. First, library code can use application services, relying on the user to supply customized facilities such as error handlers. Second, a library can use imported symbols to let an application's routines replace its own versions. Third, imported symbols give library developers the freedom to choose shared and unshared routines, without worrying about secondary dependencies. Selected routines can be excluded from a shared library, even though they are needed by other library routines.

4.1 Imported Symbol Pointers

Shared library developers can reach outside the library by defining imported symbol pointers. For example, the following code uses both an integer object and a function indirectly:

file.c

```
extern int *datum_ptr;
extern void (*print_ptr)( );
...
value = *datum_ptr;
(*print_ptr)(value);
...
```

Figure 2. Library code with indirections

Instead of referencing the imported symbols `print` and `datum` directly, the library code uses explicit indirections through `print_ptr` and `datum_ptr` to reach the desired text or data.

Depending on the number of symbols a library imports, the library code might have to change significantly. The C preprocessor and a convenient property of C allow one to hide the source code changes:

```

import.h
#define datum (*datum_ptr)
#define print (*print_ptr)

New file.c
#include "import.h"

extern int datum;
extern void print( );
...
value = datum;
print(value);
...

pointer.c
int (*datum_ptr) = 0;
void (*print_ptr)( ) = 0;

```

Figure 3. Library source code

By defining macros for `datum` and `print` in the `import.h` header file, `file.c` appears to be normal C. The compiler sees indirections, but the programmer does not.

4.2 Initialization Code

Finally, the pointers for imported symbols must be set to their proper values at run time. Libraries define their own pointers, those pointers start with a null value, and each process supplies its own initializations. Library data are private for each process; so one process's assignments cannot interfere with another's.

When the utility program builds a shared library, it knows the inter-file dependencies. Thus if an executable file needs `file.c` above, it also will need `pointer.c`. We therefore arrange for the host member for `pointer.c` to hold initialization code:

```

pointer.o
extern int datum, *datum_ptr;
extern void print( ), (*print_ptr)( );

datum_ptr = &datum;
print_ptr = &print;

```

Figure 4. Imported symbol initializations

Strictly speaking, I lied in §3.1. Some host archive members are not empty, because they hold initialization code that goes *into the executable files*. When the process starts running, it executes the initializations, thus setting all the required imported symbol pointers for the shared libraries.

This example shows the convenience of using the archive format for the host shared library. If the executable file references anything in the library that needs `datum_ptr` or `print_ptr`, the executable file will receive the initialization code for `pointer.c`. The initialization code references `datum` and `print`, forcing them to be defined too. The symbols can be avoided just as easily with the same mechanism.

I'll make one more point about imported symbols. The definitions for `datum` and `print` can reside anywhere in the process. They could be in the executable file, in another shared library, or even in the same shared library as `datum_ptr` and `print_ptr`. A library can import its

own symbols, thus supplying standard versions while allowing an application to give its own.

5. BUILDING LIBRARIES

Previous sections described how shared libraries work. This section describes how one builds a library. I mentioned a utility program that builds a shared library. Using this program, called *mkshlib*, is the primary difference between building regular libraries and shared libraries.

5.1 Library Description File

Besides the regular code needed to build a library, a library developer must control several other items: target library path name, library addresses, branch table entries, and initialization code. Developers create a separate file that specifies how to build the library. A sample specification file appears in Figure 5 for the code we've seen so far.

```
1 #target /my/directory/libX_s
2 #address .text 0x80680000
3 #address .data 0x806a0000
4 #branch
5         a           1
6         b           2
7         c           3
8 #objects
9         pointer.o
10        file.o
11 #init pointer.o
12        datum_ptr   datum
13        print_ptr    print
```

Figure 5. Library specification file

Mkshlib reads this information to build the host and target libraries. Lines 1, 2, and 3 describe where the target library should “live.” The path name must be `/my/directory/libX_s`; the text and data will reside at virtual addresses `0x80680000` and `0x806a0000`, respectively.

Lines 4 through 7 define the branch table by assigning each library function to a specific branch table entry. This order can—and should—remain constant. One can add new entries at the bottom without disturbing previous assignments. This lets a library developer expand a library without forcing application developers to rebuild old executable files. Given these lines, *mkshlib* can create the appropriate jump instructions for the branch table.

Lines 8, 9, and 10 tell what relocatable object files *mkshlib* should use to build the host and target library. The two object files listed will be linked with the branch table to form the target file, and “empty” versions of them will be collected into the host archive.

Finally, the last three lines tell *mkshlib* what initialization code to create. To the object file `pointer.o`, *mkshlib* will add the assignments shown in Figure 4.

5.2 Creating the Host and Target Libraries

With the object files and the specification file, *mkshlib* creates both the host and target library files automatically. A library developer issues one command that builds both files.

6. A CASE STUDY: THE SHARED C LIBRARY

One can build compatible versions of shared and unshared libraries. Given this possibility, one obvious candidate for conversion was the standard C library. This section describes that work.

6.1 Choosing Contents

Deciding what to include in the shared library proved to be more difficult than one might have imagined. We wanted the shared and unshared libraries to be compatible, but we did not know how much of the original library to share. We eventually based our decisions on performance and size studies.

We built a series of libraries and studied their effects on existing programs. This let us identify routines that “paid their way” and discard the “deadbeats.” For example, many existing programs use the Standard I/O package; sharing them helps many programs. Some other infrequently used C library routines define large data buffers, and we decided not to share them. Because every process gets a private copy of its libraries’ data regions, we excluded them to keep the memory requirements small.

Our resulting shared C library includes shared copies of the Standard I/O functions and other commonly used routines. Relocatable (unshared) copies of the other routines exist in the host archive—but not in the target library—for compatibility.

6.2 Redefining Library Routines

After deciding what we wanted to include in the shared library, we decided what routines we wanted to let applications replace. Although we could have allowed redefinition of every library function, that didn’t seem to be the best choice for the C library. Application writers rarely redefine most C library functions, and we did not want to add unnecessary indirections. Moreover, many redefinitions are unintentional and can cause mysterious program behavior. The draft proposed ANSI standard for C recognizes this fact by reserving library identifiers, making program behavior implementation-defined when redefinitions are present.⁵ After considering these issues, we looked at existing libraries and commands, finding what functions had previously been redefined. Fewer than 10 functions qualified.

6.3 Program Size

Existing programs proved to be a valuable test for the shared C library. We could easily measure file and memory size, adjusting the shared library contents as appropriate. We used the final shared C library and the regular C library to build many of the standard programs and achieved the following results:

TABLE 1. Shared C library results, 115 programs⁶

<i>Measurement</i>	<i>With Regular C Library (KB)</i>	<i>With Shared C Library (KB)</i>	<i>Savings (KB)</i>
File text size (avg)	19.5	8.9	10.6
File data size (avg)	5.2	4.4	.8
File bss size (avg)	4.4	4.4	0
Library text size		28.1	
Library data size		1.4	
Library bss size		0	
Memory text size (avg)	19.5	$8.9 + x$	$10.6 - x$
Memory data size (avg)	9.6	10.2	-.6
Disk file size (avg)	24.8	13.6	11.2
Disk file size (total)	2,850.5	1,559.1	1,291.4

5. ANSI, *Draft Proposed American National Standard for Information Systems—Programming Language C*, X3J11/86-017 (February 14, 1986), 73.

6. “Bss” means uninitialized data. Because the data are uninitialized they occupy no space in the executable file, even though they have memory allocated at run time.

When using shared libraries, all attached processes share the cost of library text. Thus the real memory consumption depends on system load, the number of active processes, library size, and so on. If one process is active, it must “pay” for all the memory holding the shared library text region; if 100 processes are active, the cost per process is much lower. (The “+ x” and “- x” in the table account for this.)

6.4 Program Performance

Because the shared and unshared versions of the C library were compatible, we built two versions of performance benchmark programs. Our results showed about equal performance for the two systems.

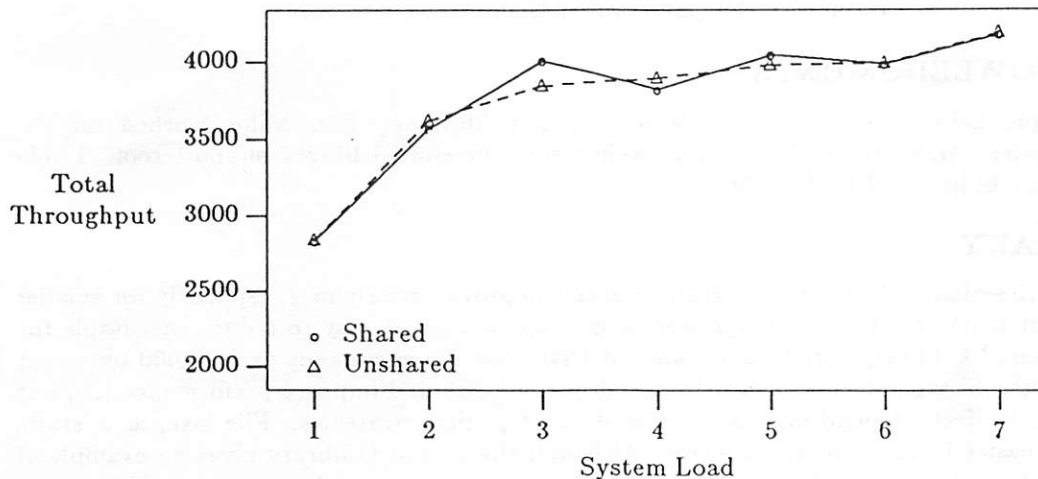


Figure 6. System performance

In the performance graph the vertical axis, “Total Throughput,” shows how much work the system accomplishes in a given time period. Higher numbers mean the system did more work, which is good. The horizontal axis, “System Load,” shows how much work the benchmarks try to do simultaneously. The higher the number, the heavier the load.

Throughput values for this benchmark frequently vary by 100 points from one run to the next. Although I’ve plotted averages from several runs and the graphs differ a little, the shared and unshared results are similar. Consequently, one can say the shared C library neither penalizes nor helps system performance for this benchmark, under the work loads tested. Although we believe this benchmark and its work load represent a fair performance test for the shared C library, they do not necessarily apply to other shared libraries or work loads.

Libraries’ static and dynamic properties can differ markedly. Some libraries (or parts of libraries) may be needed statically by many applications but executed infrequently. Error routines, for example, might reside in every executable file of a system, but they might never run if the hardware and software are reliable. On the other hand a library might have some code that only a few processes use, but the code might account for much of the total application run time.

With the shared C library, we have seen both properties. Some parts of the library execute frequently, while other routines execute hardly at all. Moreover, the C library frequently shows random execution patterns, because many different kinds of applications use its diverse routines. Libraries dedicated to a single purpose, such as database or graphics services, show much different dynamic behavior than the shared C library.

Paging systems particularly show the value of grouping closely related library functions. A library’s text region serves all processes that use the library. In the shared C library case, those processes are largely unrelated, and they touch library pages in unpredictable ways. Analyzing

the paging behavior of a shared library text region can therefore be difficult. The job becomes easier for more controlled environments. If a set of known processes uses a library of coherent functions, the working set can shrink, thus decreasing total paging activity and helping system performance.

7. FUTURE WORK

Some aspects of this effort need further work. First, statically allocating virtual addresses does not work well on machines with restricted virtual address spaces. Second, some applications want to select libraries at run time, instead of link edit time. Third, we want to add more flexibility to target path name specification. And fourth, we'll continue to enhance debugging and make the facilities available for languages other than C.

8. ACKNOWLEDGMENTS

Several people helped design and implement shared libraries. Don Milos worked on the operating system; Deborah Bach and Ken Stein wrote the shared library support tools; Iraida Torres-Irizarry built the shared C library.

9. SUMMARY

Shared libraries often help reduce program size and improve performance, especially for smaller systems. Our main goal for the design was to give applications a way to reduce executable file size. The shared C library shrank our standard UNIX System commands over 11,000 bytes per file on average. Shared libraries can help shrink process size and improve performance too, but these dynamic effects depend on system activity and paging strategies. File size, as a static property, is easier to measure and control. Although the shared C library gives an example of the possibilities, it represents a "worst case" in many ways. Applications with many cooperating processes and heavily used libraries can achieve even better results.

Decreasing Realtime Process Dispatch Latency Through Kernel Preemption

David C. Lennert

Hewlett-Packard Company
Information Technology Group
hplabs!hpda!davel

ABSTRACT

A key measure of a realtime system is how quickly a waiting process can be dispatched in response to some event (for example, I/O completion). One major component of this is the time it takes to preempt the currently executing process. In a traditional UNIX[†] system, a process executing in user code can be preempted immediately. However, when executing in the kernel, the process gives up the CPU only voluntarily and explicitly (for example, by blocking for some unavailable resource or by completing a system call). The kernel can therefore execute for a significant period of time before giving up the processor to another process. This period of time is called preemption latency and, when significant, it is unacceptable in a realtime system. This paper describes modifications to the HP-UX kernel which substantially reduce this time. Measurement results are presented which quantify these times and the improvements that have been made.

1. INTRODUCTION

This paper discusses one aspect of realtime operating system performance, process preemption latency, and presents changes made in the HP-UX operating system for the Hewlett-Packard Precision Architecture which reduce this latency.

First a brief overview of realtime system concerns and features is given. Then the process preemption latency problem is defined and alternative solutions are discussed. Finally an overview of the chosen solution and its implementation are presented followed by measurement results which quantify the improvement made.

2. REALTIME SYSTEMS

2.1. Realtime system concerns

A realtime operating system distinguishes itself from a non-realtime operating system in that it responds to a real world event within a real world time constraint. This is usually defined in terms of *event response time* and/or *sustained data throughput*.

Quick event response time allows a process to respond to an event fast enough to satisfy some external requirement. One example would be repositioning a cursor in response to a moving mouse fast enough to appear instantaneous to a human observer. Another example would be halting the supply of steel to an automated assembly line when a jam occurs.

[†] UNIX is a trademark of AT&T.

A large sustained data throughput allows quickly generated data to be gathered. If the process cannot gather data as fast as the external source is generating it, then data loss usually results. An example would be digitizing a map.

Note that the performance requirements imposed on realtime systems usually come from the physical world. The cost of not meeting the requirements can be minor (screen jitter), major (hip deep in steel), or catastrophic (reactor meltdown). This variation in cost gives rise to a range of demands on the reliability of realtime performance (from "most of the time" to "all of the time").

2.2. Realtime system features

To address these performance needs realtime systems employ features which are usually absent from non-realtime systems.

Process priorities determine the importance of a process so that a more important (stronger priority) process will execute before a less important (weaker priority) process. Timeshare systems typically adjust a process' priority frequently while it runs. On UNIX systems a process can change from a stronger to a weaker priority with respect to another process (or vice versa) many times a second. *Non-degrading priorities* allow a process to maintain a fixed (usually stronger) priority with respect to other processes and thus obtain a constant, maximum preference.

Another factor which delays process execution is swapping or paging. If a process is swapped or paged out of memory when it needs to run then response latency is increased. Allowing *processes locked in memory* prevents this problem.

Processes usually wait while a requested I/O operation (for example, a device read) is performed. Higher data throughput can be obtained by allowing a process to overlap execution with its I/O operations. This *asynchronous I/O* capability is sometimes provided directly by the operating system or can be implemented via multiple processes communicating through a high speed mechanism such as shared memory synchronized with semaphores.

3. PREEMPTION LATENCY

3.1. The problem

Non-degrading priorities and memory locked processes give maximum preference to a realtime process which is ready to execute. There is, however, one major factor to overcome before the process can execute: If another process is currently executing then it must be preempted and the realtime process restarted.

Traditional UNIX systems can preempt a process immediately while the process is executing in user mode. If, however, the current process is executing within the kernel then it only voluntarily and explicitly gives up the processor. Specifically, it gives up the processor when either 1) the kernel calls the `sleep()` routine to suspend the current process until a needed resource becomes available, or 2) the kernel returns control to the user program at the completion of a system call thereby returning to user mode and allowing preemption to occur. The kernel can execute for a significant period of time before giving up the processor to another process. This greatly increases process preemption latency and decreases the system's ability to provide quick and predictable event response time.

3.2. Alternative solutions

The goal is to decrease the amount of time the kernel executes before it gives up the processor to a waiting higher priority realtime process. To achieve this goal there are two basic alternatives: 1) the kernel can be made to execute all of its functions more quickly or 2) the kernel can be made to tolerate interrupting its execution in deference to the waiting process (preemption).

The former is clearly a superior approach as it has the side benefit of causing the entire system to execute faster and with less kernel overhead. It can be achieved through a combination of faster hardware and algorithmic changes. In addition to algorithm changes which reduce total execution time one can shift code from the kernel into the user program. One example would be to implement the file system manipulation code in a user library and leave only the code supporting basic device access in the kernel. This allows more of the "kernel" to execute in user mode where it is readily preemptable. The problems with moving kernel algorithms into user mode are a loss of reliable security checking and a loss of atomicity of operation with respect to other processes. In addition, there is only so much kernel code that can be reasonably moved into user mode. In the final analysis, preemption latency is typically left unacceptably high. So the second alternative, increasing the preemptability of the kernel, is explored.

The problem with making the kernel arbitrarily preemptable is a loss of atomicity. Kernel data structures can be viewed as memory which is shared among all the user processes. Each process makes requests of the kernel which update this shared data. There must be a mechanism which ensures that these updates are performed atomically; otherwise, faulty operations and a system crash usually result. Simultaneous (multi-processor) and interleaved (uni-processor) data structure access is prevented either through one or more semaphores (which reduces the problem to updating shared semaphores) or by preventing even the possibility of contending access. The latter approach is usually provided by atomic hardware instructions and/or so architecting the system that such colliding accesses never happen.

The mechanism used in a traditional UNIX system is this latter approach: nothing interrupts a process while it is running in the kernel. (The exception to this, I/O interrupt processing, will be discussed later.) This implementation has too coarse a granularity. That is, the data structure "lock" can be held for a long period of time which can prevent other processes from running even if they don't access the same data structures being currently updated. Thus the lock covers more data structures and lasts for a longer period of time than is usually needed. It is this drawback which gives rise to the poor preemption latency of the UNIX system.

4. SOLUTION IMPLEMENTATION

4.1. Overview

The preferred solution is to use multiple semaphores and have each semaphore control access to an independently used data structure. No other process will access the data structures which the preempted process is using since no other process has the necessary semaphores locked. The kernel can then be immediately preempted at any point in its execution. This results in the fastest preemption time but requires that the entire kernel be modified to adhere to semaphoring conventions. Just sorting through the various data structures and assigning semaphores can be a large amount of work. This approach is typically employed in multi-processor systems. For a description of one such implementation and the effort required see [Bach84] and [Felton84].

An approach which is easier to implement is to find places in the kernel where it is already safe to preempt and only allow preemption there. (Such a "safe place" is a spot or region in kernel code where all kernel data structures are either updated and consistent or locked via semaphore.) This does not require modifying the entire kernel to conform to a new data access philosophy. It does have several drawbacks though. Rather than occurring immediately, preemption is held off until the next "safe place". Also, our experience has shown that these safe places are not found but made. It is easier, however, to make a "few" safe places than to rewrite a kernel.

Because implementation schedule was of strong importance, our solution combines both these preemption styles: there is a synchronous method which allows preemption at a

specific point during kernel execution, and an asynchronous method which allows preemption anywhere during a region of kernel execution.

4.2. Details

The synchronous method is useful when places can be identified in the kernel where data structures are either in a consistent state (i.e., between an access transaction) or all required resources are locked via some semaphoring mechanism.

The synchronous method is invoked by placing a call to the macro `KPREEMPTPOINT()` at such a safe place in the kernel. This macro merely checks a global flag, `reqkpreempt`, which indicates the presence of a higher priority realtime process which is ready to run and calls a function, `kpreempt()`, to cause a `swtch()` to the process. (The `reqkpreempt` flag is similar in function to the `runrun` flag used in typical UNIX systems to indicate that a higher priority timeshare process is ready to run.)

There is also a function variant of `KPREEMPTPOINT()` called `IFKPREEMPTPOINT()` which returns true if a pending preemption was serviced; otherwise it returns false. This is useful if lengthy algorithms need to allow preemption but, if preemption occurs, there are assumptions which may have been invalidated and now must be rechecked.

The asynchronous method is useful when preemption can be tolerated over a region of execution and synchronous polling via `KPREEMPTPOINT()` would incur unacceptable overhead (for example, large memory copies during fork, exec, or user I/O). This method is implemented via a software-generated interrupt which is recognized by hardware. Hardware causes an asynchronous transfer of control to the trap handling routine (similar to a pagefault taken inside the kernel when accessing user pages) which in turn calls `kpreempt()` to preempt the kernel.

Hardware recognition of this interrupt is controlled via `spl` levels. The routines `splpreemptok()` and `splnopreempt()` have been added to allow and disallow recognition, respectively. Both of these new `spl` levels are weaker than other `spl` levels (i.e., they do not hold off other interrupts). Also, other `spl` levels are stronger and hence imply that kernel preemption is held off. Thus, interrupt processing activity which typically runs at `spl4()` or higher automatically disables preemption. Usually the kernel runs at `splnopreempt()` (whereas before kernel preemption it used to run at `spl0()`, which no longer exists).

When a higher priority realtime process becomes runnable, kernel preemption is requested by calling `preemptkernel()`. This routine sets the `reqkpreempt` flag and generates the hardware supported interrupt. The flag and interrupt are both cleared by `swtch()` whenever it switches to a new (highest priority) process.

The now pending preemption request is serviced at the first point when either:

- a) a `KPREEMPTPOINT()` is executed (which tests the `reqkpreempt` flag),
- b) the `spl` level drops to `splpreemptok()` (which allows the pending interrupt),
- c) user mode is entered (this is just one case where the `spl` level drops to `splpreemptok()`), or
- d) `swtch()` is called (which always transfers to the highest priority runnable process).

In total, approximately 180 synchronous preemption points and 20 asynchronous preemption regions were added to the HP-UX kernel.

4.3. Limitations

There is one overriding limitation on what kernel preemption can accomplish: Kernel preemption can only preempt (suspend via `swtch()`) an operation which is being executed within a process context. It cannot preempt interrupt processing code and allow a process to execute because the UNIX system does not support this type of operation.

Therefore, all interrupt processing is implicitly considered to be of higher priority than any (realtime) process. This means that no matter how quickly preemptable one makes the kernel, if interrupt processing becomes unacceptably time consuming then timely kernel preemption cannot be achieved. So the only option is to reduce interrupt processing overhead to an acceptable level.

Note that, even if all individual interrupt servicing operations are short, kernel preemption can be held off for an arbitrarily long time by many quickly arriving (back-to-back) interrupts during heavy I/O activity. There is nothing that can be done in this situation since interrupt processing, by definition, has priority over all process execution.

In addition to the typical I/O driver code, the UNIX system allows non-I/O code to be executed in an interrupt processing context. This facility, called the *callout queue*, causes a kernel procedure to be executed at a specified time offset. The procedure is invoked from an interrupt processing context during clock interrupt servicing. This is usually done at a weaker interrupt priority than all other I/O interrupts. (See [Stra86] for a more detailed discussion of the callout queue mechanism.)

4.4. Overcoming limitations

To minimize callout queue execution overhead, a separate system process was created to provide a preemptable process context in which to execute some lengthy callout queue code. This process, the *statdaemon*, is a lightweight kernel process similar to the scheduling daemon or the pageout daemon. It waits for the *lightning bolt event*[†] and then executes a standard set of statistics gathering routines. These routines represent the lengthy portion of the *schedcpu()* function. (Among other things *schedcpu()* recomputes process priorities every second; see [Stra86] for a discussion of its operation.) A new routine, *sendlbolt()*, is now scheduled on the callout queue in place of *schedcpu()*. *Sendlbolt()* performs the quick functions of *schedcpu()* including generating the lightning bolt event.

4.5. General performance improvements

In addition to the preemption specific modifications, kernel preemption times were improved by several general performance improvements. These include making the process table multi-threaded and placing entries in different states on different lists, as well as using hashing techniques to speed data structure searches. See [Feder84] or [McKusick85] for a discussion of similar improvements.

4.6. Debug facilities

A set of debug facilities allow dynamic control over the preemption system. These facilities are conditionally compiled into the kernel and include the ability to turn preemption on or off, enable kernel preemption for timeshare as well as realtime processes, and forcing (almost) all preemption points to "preempt" (sleep for a random amount of time).

4.7. Timing measurement facilities

In order to tell how long the kernel executes without blocking or preempting, and where in the kernel these long execution paths are, the kernel was instrumented to collect timing measurements.

Timing measurements are taken by sampling the time at kernel entry and exit (*syscall()*, *trap()*, and *swtch()*) and also whenever the kernel changed between a preemptable and non-preemptable state (*KPREEMPTPOINT()* and all *spl* routines). The time intervals during which the kernel executes in a non-preemptable state are logged. Also, in order to tell where in the kernel this execution occurs, a program counter (*pc*) procedure call trace of the kernel stack is collected when the time is sampled, and the *pc* traces for start and stop points of the

[†] The lightning bolt event is a standard UNIX event which occurs frequently, for example, every second.

interval are logged along with the delta time.

However these times fluctuate based on the amount of interrupt processing activity which occurs during the measurements (since device interrupts are injected "randomly" into non-interrupt kernel execution). To account for this, the measurement system also counts the interrupt processing time which occurs during a kernel path measurement and logs this as well. This allows the interrupt processing time to be precisely removed from the "normal" kernel execution times. Results are reported both ways.

The interrupt processing time is counted in a fashion similar to the "normal" execution time. Whenever the processor switches to or from the interrupt stack the time is sampled. On exiting the interrupt stack, the time difference is computed and added into an ongoing count. This count is zeroed at the beginning and logged at the end of a "normal" measurement.

In order to obtain sufficiently accurate times, a new kernel routine was introduced to obtain clock time accurate to a microsecond.

Data logging is done into a circular kernel buffer. A user level program retrieves the data by reading the pseudo-device file, `/dev/kmem`, which accesses kernel memory.

A workload is run on a kernel equipped with the measurement system while data is collected into a file. The file is later reduced by a set of user level programs.

5. MEASURED IMPROVEMENT

5.1. Presentation of results

To determine the improvement made in realtime process dispatch time, two sets of measurements were taken using the timing measurement facilities discussed above. One set with kernel preemption enabled and one set with it disabled. The same workload was run during both measurements.

The workload consisted of a suite of tests which validate the correct working of all kernel functions. Because the instrumented kernel precisely measures each section of the kernel every time it is executed, it is not necessary to execute a kernel code path more than once.

The raw results consist of a stream of times. Each time represents an interval when the kernel was non-preemptable. As one way of summarizing the results, a distribution was computed which represents the percentage of total kernel execution time that was spent in code paths of less than x milliseconds.

Figure 1 shows this distribution for both preemption on and preemption off. Figure 2 shows a blowup of the portion of Figure 1 near the y axis and more clearly shows the "preemption on" case. Each graph can be interpreted as: y percent of kernel execution time was spent in code paths shorter than x milliseconds, or y percent of the time, the kernel can be preempted in less than x milliseconds. A sharply rising curve indicates that more of the total system time was spent in quickly preemptable code paths. It is readily apparent that without kernel preemption most of the kernel execution time occurs in code paths which are fairly long. The endpoint on each curve indicates the maximum observed value (observed worst case) for this test run. Note, however, that the curve labelled "Without Kernel Preemption" has been clipped in Figure 2; the real endpoint is shown in Figure 1.

The results in Figures 1 and 2 are summarized in Table 1.

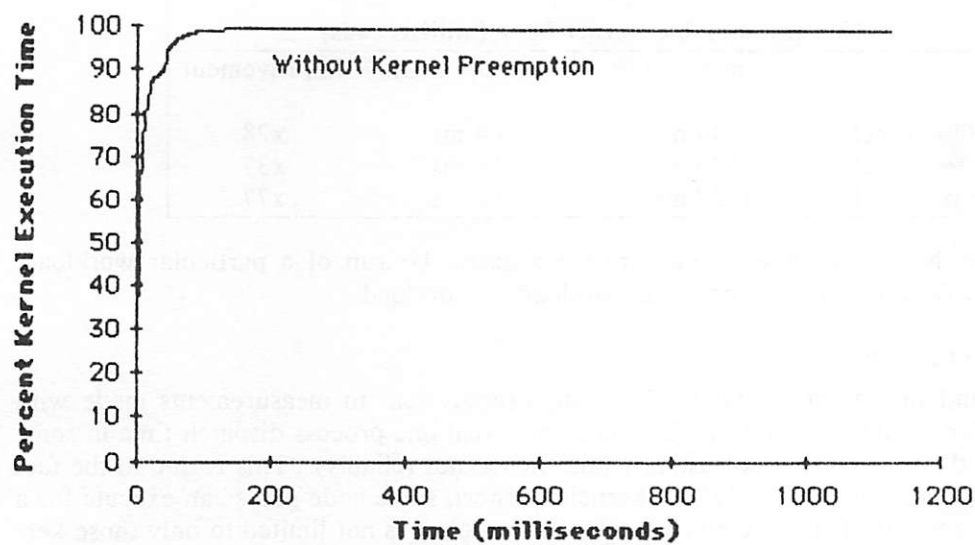


Figure 1

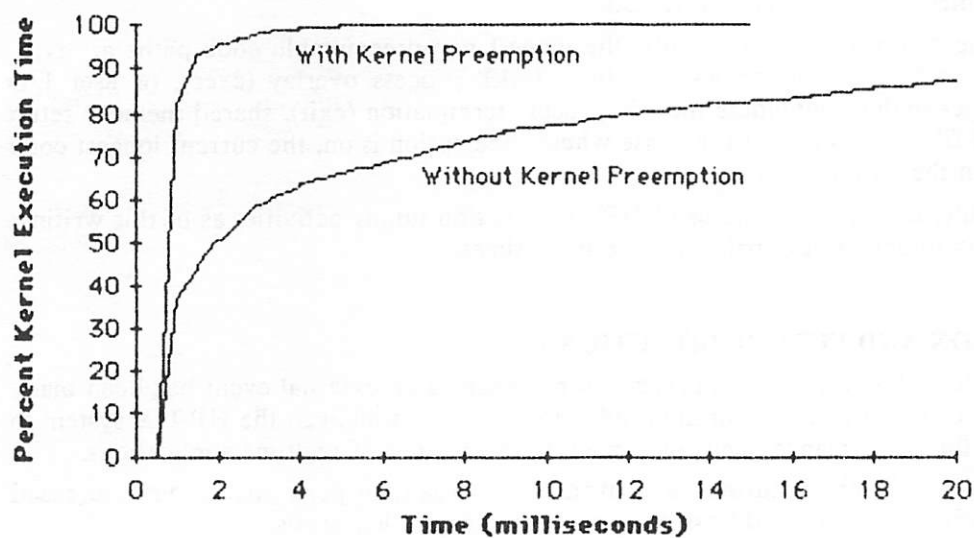


Figure 2

Table 1 Non-preemptable Kernel Time (milliseconds)			
	Preemption Off	Preemption On	Improvement
90% kernel	40 ms	1.4 ms	x28
99% kernel	129 ms	3.4 ms	x37
max kernel	1127 ms	14.6 ms	x77

Disclaimer: Note that these results are for a particular run of a particular workload. Results will vary from run to run and from workload to workload.

5.2. Discussion of results

It was found that a traditional UNIX system (equivalent to measurements made with kernel preemption disabled) could provide acceptable realtime process dispatch time in some cases but it could not provide it consistently (and hence not reliably). This is due to the fact that, while many code paths in the UNIX kernel are short, some code paths can execute for a very significant amount of time. Unfortunately, this problem is not limited to only those kernel code paths executed on behalf of the realtime application; any simultaneously running application(s) could cause the kernel to execute a lengthy code path. This means that, on a traditional UNIX system, a realtime application which works simultaneously with one background workload may fail with a different workload or even with the same workload on a different occasion.

In contrast, Table 1 shows that HP-UX kernel preemption has provided significant improvements in realtime process dispatch time. In the worst case observed with our workloads the improvement was well over 50 fold. With preemption enabled it was found that non-preemptable kernel code paths were significantly shorter, and more consistently so, than in the traditional case. This resulted in better and more reliable timely dispatch of realtime processes regardless of background workload.

In the case where preemption is off, the longest non-preemptable code paths are typically large data copies during process creation (fork), process overlay (exec), or user I/O operations. Other major contenders include process termination (exit), shared memory setup operations, and file link/unlink. In the case where preemption is on, the current longest code paths are now in the terminal driver.

These results represent the status of HP's preemption tuning activities as of this writing; work is currently underway to further reduce these times.

6. EVALUATION AND FUTURE DIRECTIONS

The time to dispatch a waiting process in response to an external event has been made significantly smaller and more uniformly predictable. This has allowed the HP-UX system to solidly address the performance needs of a much broader range of realtime applications.

Future work will entail further improvements to realtime performance via increased kernel semaphoring in order to address more stringent application needs.

7. ACKNOWLEDGEMENTS

The following people from Hewlett-Packard contributed to this work: James O. Hays introduced preemption points and regions into the memory and process management systems; he also offloaded interrupt processing functions into the newly created statdaemon. Sol F. Kavy introduced preemption points and regions into the file system. Suzanne M. Doughty edited early versions of this paper.

8. REFERENCES

- [Bach84] M. J. Bach and S. J. Buroff, "Multiprocessor UNIX Operating Systems", *AT&T Bell Lab. Tech. J.*, **63**, No. 8 (October 1984), pp. 1733-1749.
- [Feder84] J. Feder, "The Evolution of UNIX System Performance", *AT&T Bell Lab. Tech. J.*, **63**, No. 8 (October 1984), pp. 1791-1814.
- [Felton84] W. A. Felton, et. al., "A UNIX System Implementation for System/370", *AT&T Bell Lab. Tech. J.*, **63**, No. 8 (October 1984), pp. 1751-1767.
- [McKusick85] M. Kirk McKusick and Mike Karels, "Performance Improvements and Functional Enhancements in 4.3BSD", *USENIX Conference Proceedings*, Summer 1985, pp. 519-531.
- [Stra86] Jeffrey H. Straathof, Ashok K. Thareja, Ashok K. Agrawala, "UNIX Scheduling for Large Systems", *USENIX Conference Proceedings*, Winter 1986, pp. 111-139.

MOS – Scaling Up UNIX

Amnon Barak
On G. Paradise

Department of Computer Science
The Hebrew University of Jerusalem
Jerusalem 91904, Israel
amnon%Israel@csnet-relay

ABSTRACT

MOS is a Multicomputer Operating System which integrates a cluster of loosely coupled, autonomous and homogeneous computers into a single-machine UNIX environment. Its main properties are network transparency, decentralized control and load balancing by dynamic process migration. The internal structure and the algorithms of MOS are designed to allow scaling-up of the system to a large number of computers. This paper describes the difficulties of building a large scale multicomputer system and discusses some algorithms used to overcome these problems in MOS.

Introduction

A large scale UNIX[†] system can be more cost effective than a number of small ones. Due to the economy of scale, computing resources can be shared and need not be replicated. A single, big system is also easier to maintain and provides better facilities for information exchange within the user community.

Mainframe computers [1], however, are less cost-effective than smaller machines, because they are closer to the limits of current technology and are not mass produced. Departmental computers and personal workstations also provide more autonomy, and hence better uptime, to their users.

A distributed UNIX built out of hundreds of small computers might enjoy the best of both worlds. Sharing of information and disk space is provided by a distributed file system with a uniform name space. Other resources, such as CPU cycles, are harder to share, but this can be overcome by techniques of load balancing.

Distributed systems might seem to be extendible *ad infinitum*: network technologies can easily support systems with hundreds of computers. However, it is naive to assume that a hundred processor UNIX would provide a hundred times the throughput of a single processor system. In this paper, we explore some of the difficulties of building a large scale distributed operating system and then discuss the properties which make MOS scalable to a many machine configuration.

What is MOS?

To the programmer, MOS [2] is a relatively large, conventional UNIX environment. The hardware architecture is a cluster of identical microcomputers (PCS/Cadmus 9000, with a MC68010 CPU and a megabyte of memory). Each computer has its own disk and other peripherals and may function independently. A medium speed local area network (Proteon 10Mbps, token-passing Pronet) ties the machines together.

The MOS kernel is divided into two levels. The lower part is tightly coupled with the hardware and is site-sensitive, while the upper part has no notion of the particular processor it is running on. User processes interface only with the upper part of the kernel, which provides a classic UNIX interface. Therefore, processes are site-insensitive and can migrate within the cluster, to make use of idle CPU cycles.

[†] UNIX is a trademark of Bell Laboratories.

Some Internals

The communication between the upper and the lower kernel is done via a remote procedure call mechanism. Whenever possible, information is transferred as *inter-machine pointers*. For example, all references to *inodes* within the upper part of the kernel is done via *universal inode pointers*:

```
struct u_inode {
    struct inode *ui_p;          /* inode in lower kernel */
    short ui_machine;           /* lower kernel machine number */
    short ui_version;           /* unique stamp */
};
```

As a result, the *namei* primitive (that converts file names to *inode* pointers) can be split between the two parts of the kernel.

```
/*
 * Convert file name to universal inode -- Upper part
 */
struct u_inode
namei(path, flag)
char *path;
{
    struct u_fname component;
    struct u_inode ui = (*path == '/') ? u.u_rdir : u.u_cdir;
    /* start from root if path begins with a '/' */

    while (/* path is not exhausted */) {
        /* Copy first component of path into component */
        ...
        /* RPC to get the inode of component */
        ui = Scimb(ui.ui_machine, ui, component);
    }
    return(ui);
}
```

The call to *Scimb* is passed as an RPC to the lower part of the kernel in machine *ui.ui_machine*. This sets a lightweight process, called *ambassador*, to represent the calling process at the remote machine. The ambassador invokes the lower-kernel function

```
/*
 * Climb one step on the UNIX file system tree.
 */
struct u_inode
climb(ui, component)
struct u_inode ui;
struct u_fname component;
```

which scans the directory pointed to by *ui* and returns the (universal) inode of the file named *component*. An ambassador never runs in user mode. As soon as *climb* returns, it sends the result to the invoking upper kernel, and goes back to sleep until another RPC is received.

The mechanism above provides all processes with access to the conventional UNIX file system which is maintained by each lower kernel.

Data transfers between the user address space and the lower kernel are done via a *funnel*, which is another type of universal pointer. A funnel is set by the upper kernel to point into the process address space and to maintain a count of the transferred data. The lower kernel uses this funnel to copy data across the network at a rate similar to a memory-to-memory copy.

Efficient inter-processor mechanisms are only one aspect of a distributed, large scale system. The following sections address some of the other difficulties and discuss the solutions implemented in MOS.

Fault Tolerance and Isolation

MOS never uses more than two machines to carry out a single transaction: a system call, for example, is served by one upper kernel and one lower kernel. That way, the probability that one of the participating machines will fail is only twice as great as in a single-processor system.

As the number of processors increases, more of them are expected to fail while the others keep running. Unless the system detects single processor faults and reconfigures itself accordingly, such a fault might degenerate into a global failure. Once such a fault is detected, some measure of dynamic system reconfiguration must take place.

Failure of a single processor effects remote processes which use its resources, as well as processes running locally. The handling of the first case is relatively simple: the remote processes receive a signal and may continue to run if the lost resource was not essential. In the second case, processes that were running on the faulty processor die and may leave behind allocated resources such as open files, in-core *inodes* and ambassadors. These *orphan* resources will be lost unless the system identifies and reclaims them gracefully.

In MOS, when an object is allocated, a time-bomb is attached to it. Processes are responsible for resetting the timer. This is done whenever a object is accessed, or when it receives a special *keep alive* message. A garbage collection process scans all of the objects and reclaims those whose timers have expired. After an object has been garbage collected, it is possible that inter-machine pointers still point to it. To prevent access to expired objects, each object is tagged with a unique *version* number upon its allocation. This version number becomes a part of each of the *universal pointers* to the object. Whenever a universal pointer is converted into a local one, a consistency check is made to ensure that the version numbers match.

The above algorithm may seem complex and costly. Since the probability of a single machine failure is quite low, the algorithm is run infrequently. For example, keep alive messages are sent every minute and the garbage collector is run every ten.

Dedicated Servers

Another pitfall that might impair the performance of a scaled-up system is usage of a non-distributed, dedicated server. Failure of such a server might cause a large portion of the system to fail as well (as pointed out by Popek *et al* in [3]). Additionally, loading a server to saturation might result in a global system congestion: processes that wait for an overloaded server tie up system resources such as kernel data structures, swap space and keep-alive messages.

In order to be scalable, all server mechanisms are distributed. MOS servers and clients share the same processors, which has the desirable side effect of delaying would be clients from sending their requests to loaded servers. As a server becomes loaded, it consumes more resources and leaves less for its would be clients. This is especially noticeable if clients are dynamically relocated to the processor on which their request is processed.

Overhead of Scaling

The UNIX kernel assumes that information about the global state of the system is available to all processes. Some services that rely on this information become more expensive as the system is made larger. Consider, for example, sending a signal to a process group (*killpg*). In a single processor system with p processes, this involves scanning the *proc* table, which costs $O(p)$ operations. In a n -node system, not only might there be $n \times p$ processes, but the number of calls to *killpg* is likely to grow as well. The system-wide cost now becomes $O(n^2p)$, which means n times more work for each processor.

An ideal scalable system should keep the cost of handling an object constant, regardless of the total number of objects. Some UNIX primitives (e.g. *namei*), have overhead of $O(\log n)$, which is quite acceptable. However, conventional methods for handling global system information usually introduce an overhead of $O(n)$ or worse, which is intolerable.

Since global information is expensive, if not impossible, to maintain [4], MOS tries to make the most out of the partial information that is available at each site. As a rule, it is preferable to approximate the optimal action using partial data rather than to pursue a truly optimal solution based on global information. In this spirit, MOS makes extensive use of information scattering and non-orthodox, probabilistic algorithms. Some of these are described in the following sections.

Load Balancing and Information Scattering

The load-balancing algorithm of MOS consists of three parts. The local *load monitor* measures the load of each processor; the *exchange* algorithm routes load information throughout the system; the *process migration* algorithm dynamically migrates processes at any stage of their execution from overloaded to underloaded processors.

In order to overcome the failure of any site, the load exchange algorithm uses *random information scattering*, which is immune to partial failures of the system. Other advantages of this strategy include low communication overhead, since no network broadcasts are used, and decentralization, since all the nodes use the same algorithm. No synchronization is required between the nodes. In spite of these loose requirements, the scheme is extremely efficient: as shown in [5], information can be transmitted from any node to all of the others in $C \log n$ units of time, where $C < 2$.

The load scattering algorithm works as follows: at fixed intervals, each node randomly selects one other node and transmits the local load information to it. Each node maintains information about the load of a small subset of the other nodes. This information is updated frequently, as new messages are received. This way, each MOS site knows the load of a random subset of the others, those which have selected that site in the last few units of time. It may choose to balance its load by migrating processes to a node of this subset. Note that no global information is maintained: the local load table of each node is small and contains information about only a part of the system.

If a node is overloaded and the information that it has about other nodes does not justify load balancing, then, by default, this node need do nothing. It simply waits until the information exchange algorithm provides it with information about some other nodes. An underloaded node advertises its state to some subset of the system. As a result, it is approached by a few other nodes and will not be overwhelmed by load balancing from some large group of overloaded nodes.

As a whole, the load balancing algorithm is geared to provide the best performance within the frame of information available to each node. The overhead of migrating a process, comparable to swapping it out, should be weighed against the potential gain. The load balancing mechanism is tuned to optimize this gain and to avoid back and forth process migration. The implementation also includes optimization of I/O requests. Processes that require heavy I/O are migrated to the site where the remote I/O operations take place. More about the load balancing algorithms and some performance results are given in [6].

Process Locating

As noted above, MOS processes migrate freely from one site to another. The UNIX signal mechanism requires that a migrated process can be easily located. For that purpose, each process has *homes*, which point to its last known location.

A hash function converts a process number (*pid*) to a set of machines where its homes are created. The number of homes is tuned by probabilistic assumptions about the rate of node failure, so that with high probability at least one home will survive a major crash. In order to locate a process, one need only use the hash function (which is known to all the kernels) to locate all of the homes of the process. It is easy to verify that if a process and at least one of its homes survive a crash, the process can be located. This scheme allow processes to reestablish homes in nodes which recovered from a crash.

Homes are a consumable resource and are subject to garbage collection. As with orphan objects, each process is responsible for keeping its homes alive, by sending keep-alive messages.

Conclusion

Experience with the MOS system has provided the following guidelines for maintaining scalability:

- Inter-processor mechanisms can be made comparable in efficiency to conventional UNIX. This efficiency is essential, but not sufficient for scalable systems.

- As single machines can fail, they will. Probabilistic algorithms provide some measure of immunity against this. Limiting the life time of objects is also useful.
- Handling global information is of intolerable complexity. On a large scale system, this information might be meaningless: what would be the meaning of *who* or *ps a* on a hundred machines?
- Optimal solutions are costly. In many cases, suboptimal solutions are sufficient.

MOS has been fully operational for over two years. Besides ongoing research in reliability, performance modeling, parallel algorithms and distributed applications, it provides computing power for our operating system courses.

In general-purpose computing, the performance of MOS compares favorably with that of System V (release 2), even when process-migration is disabled. Migrating a process is comparable in cost to swapping it to disk and it improves response time significantly.

There are a few applications for which a speedup of $O(n)$ has been attained by distributing them among n processes. Such a speedup is typical for CPU-bound problems which can be split into sub-processes that share a very small amount of data.

Some mechanisms found in the eighth edition of UNIX would suit MOS very well. The concept of *streams* [7] may be integrated with that of *funnels* to provide efficient inter-process communication [8]. Addressing processes as files [9] may take advantage of the MOS unified file system.

Acknowledgements

The material presented in this paper is based on research supported in part by the United States Air Force Office of Scientific Research under grant AFOSR-85-0284. Partial support has also been provided by the Israeli National Council for Research and Development.

MOS was ported to the 68000 by Amnon Shiloh, who actually re-wrote the PDP11/23 version. He should be credited for many of the ideas in this paper.

Rick Wheeler made this paper human-readable.

References

1. T. W. Hoel and B. J. Keller, A Unix-based Operating System for the Cray 2, in *Usenix Technical Conference Proc.*, Denver, Co., Winter 1986, 219-224.
2. A. Barak and A. Litman, MOS: A Multicomputer Distributed Operating System, *Software Practice & Experience* 15, 8 (Aug. 1985), 725-737.
3. G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin and G. Theil, Locus – A Network Transparent, High Reliability Distributed System, *Proc of the 8th SOPS* 15, 5 (Dec. 1981), 169-177.
4. B. W. Lampson, Applications and Protocols, in *Distributed Systems – Architecture and Implementation*, B. W. Lampson (ed.), Springer-Verlag, Berlin, Heidelberg, 1981, 357-370.
5. A. Barak and Z. Drezner, A Probabilistic Algorithm for Scattering Information in a Multicomputer System, CRL-TR-15-84, University of Michigan, Ann Arbor, Mi., March 1984.
6. A. Barak and A. Shiloh, A Distributed Load-balancing Policy for a Multicomputer, *Software Practice & Experience* 15, 9 (Sept. 1985), 901-913.
7. D. M. Ritchie, A Stream Input-Output System, *Bell Sys. Tech. Jour* 63, 8 (October 1984), 1897-1910.
8. D. M. Ritchie and D. L. Presotto, Interprocess Communication on the Eighth Edition Unix System, in *Usenix Conference Proc.*, Portland, Or., Summer 1985, 309-316.
9. T. J. Killian, Processes as Files, in *Usenix Conference Proc.*, Salt Lake City, Utah, Summer 1984, 203-207.

A Data-Flow Manager for an Interactive Programming Environment

Paul E. Haeberli

Silicon Graphics Inc.
2011 Stierlin Road
Mountain View, California 94043

EXTENDED ABSTRACT

ABSTRACT

Multiple windows are a common feature of contemporary interactive programming and application environments, but facilities for communicating data between windows have been limited. This paper describes extensions to an operating system that allow graphics programs to be combined in a flexible way. A *data-flow manager* is introduced to control the flow of data between concurrent processes. This system allows the interconnection of processes to be changed interactively, and places no limitations on the structure of process interconnection. As a result, this environment encourages creation of simple, modular graphics tools that work well together.

1. Introduction

Although multi-window systems [10] are popular, these environments provide only a *glimpse* at the real potential of the current generation of graphic display hardware. Unfortunately, advances in systems software have lagged far behind advances in computing and display hardware. One of the most popular operating system in use today is UNIX*, a system whose user interface is better suited to text processing on ASCII terminals than more general data processing on the current generation of graphics displays.

The work described here is motivated by the growing complexity of interactive programs operating on the IRIS [7] workstation, and the inadequacy of the UNIX operating system in supporting our needs. Our goal is to develop an interactive graphics development and applications environment that is not limited by the facilities of UNIX.

2. Synergy

One common problem in our environment is viewing a geometric object interactively. Significant work is needed to parse user input, and generate proper views of objects. Additional interaction may be desired to allow a sequence of views to be saved and replayed, or to allow control over illumination direction, or other global properties. The thought of additional, even more complex interaction handlers motivated the development of an environment where simple tools could be applied in a variety of ways.

* UNIX is a Trademark of Bell Laboratories.

Synergy describes a system whose capabilities are unpredicted by the capabilities of each of its components taken alone. Modern computer systems promote synergy to provide powerful programming environments. This can be done by encouraging development of simple, *modular* tools that can be *combined* in a flexible way to build information processing functions.

3. The Unix Environment

UNIX is one example of a synergistic system. It consists of a set of conceptually simple, *modular* tools (programs) that can be combined in a flexible way to create various information processing functions. Each primitive tool has one input port and one output port. These primitive tools may be combined by *pip-ing* the output of one into another.

The command interpreter allows the structure of the connection to be given on the command line with a syntax like:

```
% ls -s | sort -r | head -10
```

This command lists the names and sizes of the files in the current directory (`ls -s`), then sorts this output in reverse numerical order (`sort -r`), and displays the first ten lines (`head -10`). As a result this command line construction lists the 10 largest files in the current directory.

The UNIX system encourages creation of simple tools that do one thing well. Combining tools is easy, and the power of the system exceeds that of the tools taken separately. This is one of the important strengths of UNIX.

The UNIX model is nicely integrated with the ASCII terminal. The user's key strokes and the ASCII display are viewed as streams of characters. The interprocess communication (IPC) facility is suited to many tasks, such as text processing or general transformation of a single stream of data. It is adequate for any information processing task that can map onto a pipeline of transformations as shown in Figure 1.

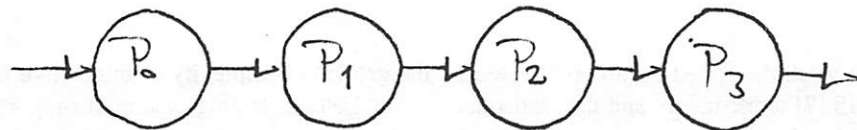


Figure 1. UNIX interprocess communication

Unfortunately, there are severe limitations inherent in the UNIX model. Each process has only one input and one output, and the UNIX command interpreter supports only the construction of a pipe of processes, one sending data to the next. UNIX does not provide the ability to connect processes with edges of an arbitrary graph. Further, the interconnection of a group of processes can not be changed dynamically, and the user can interact only with one process in the pipeline. These are serious limitations in a multi-window environment.

The current generation of graphic display systems differ from the conventional ASCII terminal in many important ways. One difference is the quantity of information provided by the display. A typical ASCII terminal displays about 2,000 bytes, with an output bandwidth of 2,000 bytes per second. Compare

this to a graphics system with about 1,000,000 bytes in the display and an output bandwidth of 1,000,000 bytes per second for pixels or geometric objects!

Another difference is the dimensionality of the display medium. The ASCII terminal displays a one dimensional stream of characters. A graphics display is capable of displaying two and three dimensional objects. While a one dimensional pipe is adequate for an ASCII display, a multi-window environment needs the ability to connect processes in two dimensions.

4. The Data-Flow Environment

In the data-flow environment described in this paper each process is given up to eight input ports and eight output ports. A data-flow manager allows the user to connect an output port of one process to an input port of any other process. These connections may be changed as processes are running. In addition, any output port may send data to more than one input port, and a single input port may receive data from more than one source. The process interconnection can be described by a general directed graph with processes as vertices, and connections as edges. This is shown in Figure 2.

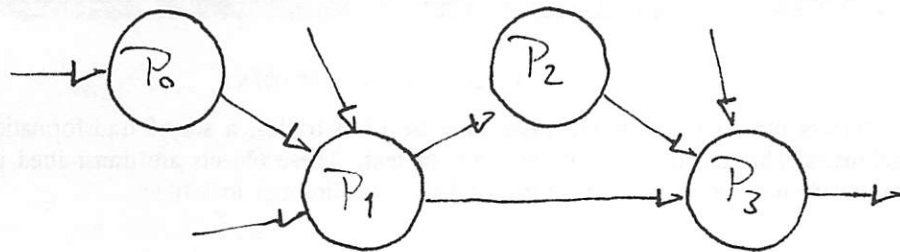


Figure 2. General Data-Flow

In a typical application we may want to control the orientation of a geometric model for viewing, record a series of views, and play back the resulting sequence. To do this three simple tools are used: an object viewer, a view editor, and a recorder.

First the individual tools are started by making selections on a menu. The object viewer displays an image of the geometric object; it has two input ports, an object input and a transformation input. The view editor presents a set of sliders to control rotation, translation and scale, and has a single transformation output port. The stream recorder may be used to record a sequence of views and play these back in a loop; it has one input port and one output port.

After the tools are started, the data-flow manager is used to connect the output of the view editor to the view input of the object viewer, and to the recorder. Another connection is made between the output of the recorder and the object viewer. The screen image of this network is shown in Figure 3.

Now as any slider on the view editor is moved, the object viewer updates its image of the object. We can also put the recorder into record mode, slide the sliders about while viewing the motion, and play back this set of views to the object viewer. It is easy to create simple animated views in this way. It is also easy to make a second object viewer use the same orientation as the first by simply creating a connection.

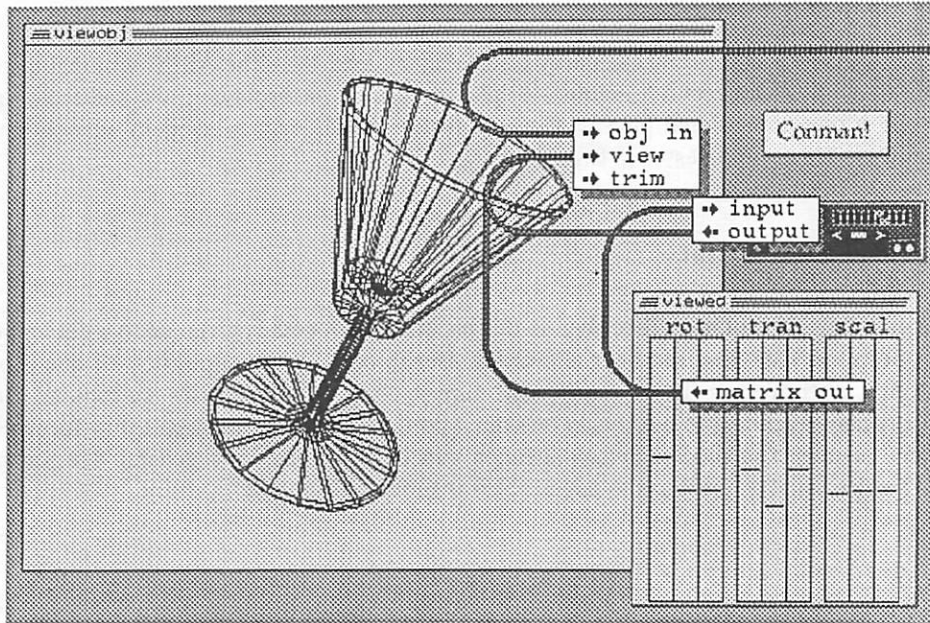


Figure 3. Viewing an object

The objects passed between processes may be RGB triples, a set of transformation commands, images, geometric objects, fonts, stipple patterns, or text. These objects are transmitted in ASCII form whenever convenient. For example the output of the view editor may look like:

```
rotate 60.53 x
rotate 23.95 y
rotate 0.0 z
translate 2.32 0.0 1.23
scale 1.0 1.0 1.0
```

Figure 4. Output of the view editor

Other applications are easy to build out of data-flow components. This system can be used to connect the output of a font editor to a text window to preview a font, or allow a shape editor to modify the brush being used by a paint program, or transfer text from one window to another. These data-flow networks are easily created and modified with the data-flow manager.

5. Applications

Several data-flow tools have been developed that may be applied to any type of object that is transmitted in ASCII form such as transformations, geometric objects, etc. A recorder process allows a sequence of objects to be recorded and replayed. This may be used to create an animated sequence of views, or geometric objects. An interpolation tool has two input ports and one output port. Moving the mouse causes the output to change from one input to the other. This tool can interpolate between two views, between two geometric objects of similar structure, or between two RGB values. Other tools include a first order lowpass filter, and a filer. It is easy to think of additional tools to provide support for key frame animation.

Some object-specific tools also have been developed. A program called sweep allows a transformation to be applied repeatedly to a complex polygon to create surfaces of revolution, as well as surfaces of extrusion. This is shown in Figure 5. Line drawings may be entered with a shape editor that outputs a geometric object. A graftal [9] plant tool can be sent a viewing transformation, a leaf object (possibly from the shape editor), and a gene. An application of this tool is shown in Figure 6.

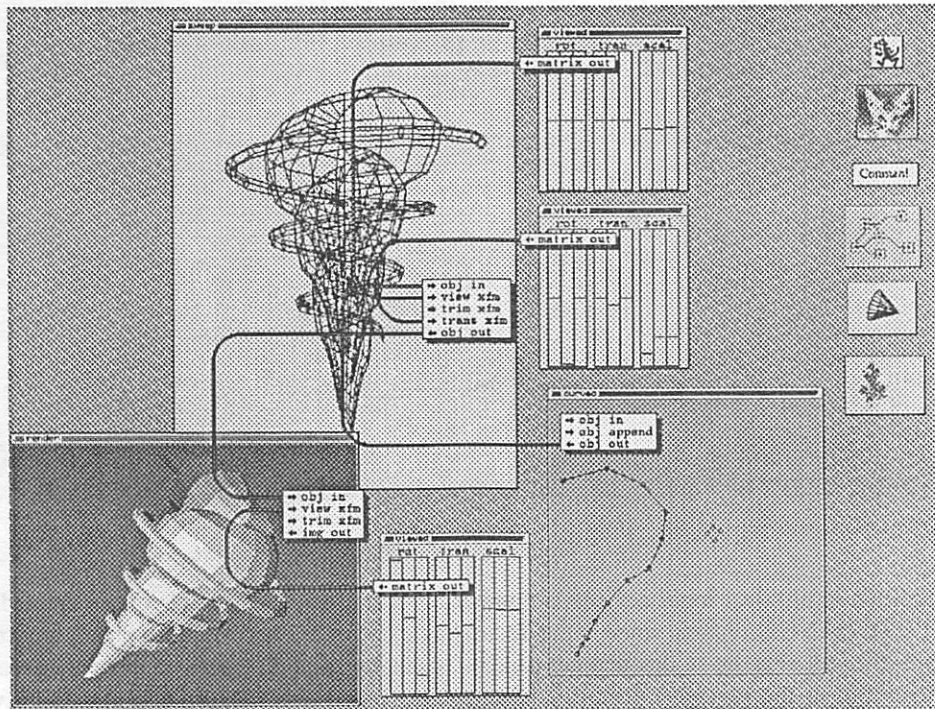


Figure 5. A surface design application

6. Interprocess Communication (IPC)

The communication between processes is accomplished by typed, variable sized, synchronous messages. Messages are multi-cast if a given output is sending to more than one input. The sender is blocked until all receiving processes have consumed the message. This IPC mechanism uses an input event queue associated with each process.

Input Events

An input queue is supported by the IRIS graphics library [8]. Each event in the queue consists of a device number and a value. Individual events are read using a statement like `dev = qread(&value)`. This call will suspend the process if there is no data in the queue. Presence of data in the queue may be tested with the function `qtest()`.

A process may request that events from a device be added to the queue with a command of the form `qdevice(dev)`. If the device is a button, like `LEFTMOUSE`, an event will be added to the queue each time the button changes state. The value will be either 0 for up or 1 for down. If the device is a valuator, like `MOUSEX`, events will be added to the queue as the valuator changes.

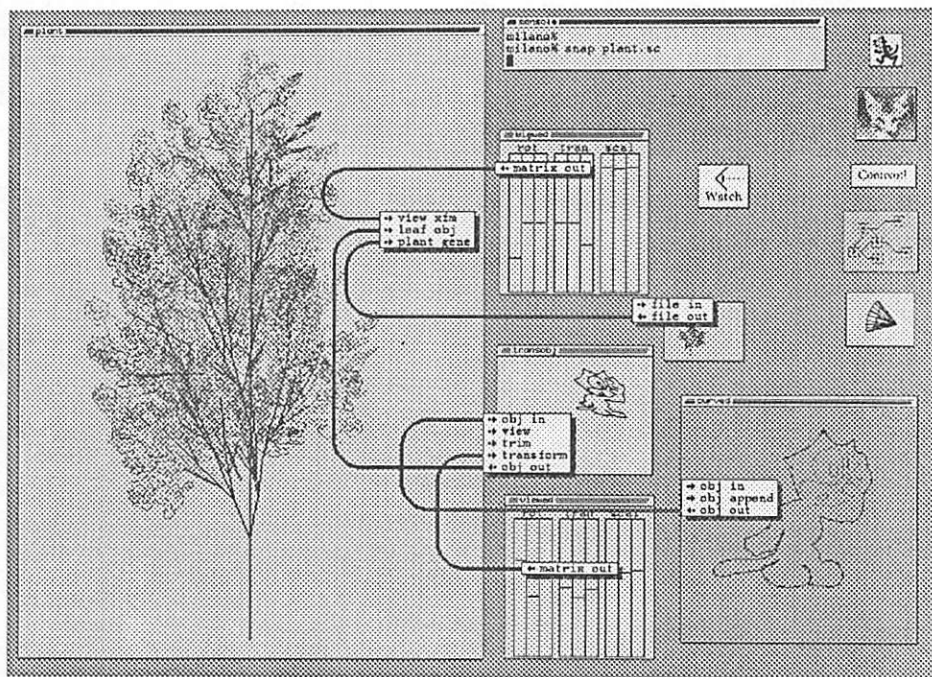


Figure 6. A plant design system

In addition to physical input devices, a number of virtual devices have also been created. A device called **INPUTCHANGE** reports when input focus has changed. Other virtual devices report when any window is destroyed, or indicate that another process is sending us a message. In this way **qread(&value)** may be thought of as the **receive()** primitive of a message based operating system [3]. Eight virtual devices **INPUT0** through **INPUT7** are used to notify a process that new data is available on an input port.

The Kernel

Interprocess communication is synchronized by the kernel, while actual data is communicated through the file system.

The kernel maintains a linked list of connections associated with each output port of every graphics process. This list may be null if an output port is not connected to anything. Each element of the list establishes a connection between that output port, and an input port of another process. Each connection element has three fields: a pointer to the next connection element; a process identifier; and an input port number.

Several system calls have been added to UNIX in addition to the **qdevice**, **qtest**, and **qread** system calls described above. These new kernel functions are shown in Figure 7. The implementation of these system calls requires about 200 lines of C code.

Connections can be thought of as dependencies. A connection between an output port of one process and an input port of another process makes the input port of the second process dependent on the output port of the first process. **Makecon** and **breakcon** are used to add or delete connections (dependencies).

```

makecon(srcpid,srcport,dstpid,dstport)
breakcon(srcpid,srcport,dstpid,dstport)
sigport(srcport,val)
replyport(srcport)

```

Figure 7. Kernel Functions

Sigport(outputport,val) is used to send a signal to all processes that are dependent on the specified output port. An event is added to the input queue of each dependent processes. The value **val** is the value they will see as a result of **qread(&value)**. The **sigport** system call suspends the sending process until all dependent processes have replied with **replyport(dstport)**. This **sigport**, **qread**, **replyport** sequence forces synchronization on messages.

Communication of Data

Communication of actual data is accomplished in the following way. The sender writes its output to a file in a well known directory. The actual filename is constructed by concatenating the string "ipc" and a four digit number composed of the sender's graphics process id, and the output port number. Then the sender uses the function **sigport(srcport,val)** to indicate that new data is available from this process.

Each process connected to this output port will receive an entry in its input queue indicating that there is new data available on one of its input ports. The integer value of the event provides the four digit number indicating where to find data associated with this event.

The functions **sigport** and **replyport** are hidden from applications by a higher-level interface. Two functions **openipc(portno)** and **closeipc(f)** are used by senders and receivers. **Openipc(portno)** opens the appropriate file for reading or writing, and returns a pointer to a UNIX stream. This makes it easy for applications to use the standard i/o library to read and write data. **Closeipc(f)** flushes output to the file and calls **sigport(srcport,val)** or **replyport(srcport)** as needed.

In practice **fprintf** and **fscanf** were found to be extremely slow for the transmission of floating point numbers. By optimizing some floating point conversions and creating specific functions to get and put floating point numbers, throughput was improved by a factor of ten.

A library of functions has been developed to make it easy for applications to read and write various types of objects. These functions convert from data structures in memory to an ASCII representation and back. The object types currently supported are geometric objects, RGB colors, transformations, and graftal genes.

7. The Data-Flow Manager

The Data-Flow Manager or connection manager (**conman**) is a user process running under the window manager. **Conman** supports the needs of the user and applications in the data-flow environment. Client processes need to provide text labels for their input and output ports when they start running. The user needs to be able to alter the interconnection of processes.

When **conman** starts up, it registers itself as *the* unique connection manager. This allows clients to communicate with **conman**. When a client starts it sends messages to **conman** indicating the input and output ports it uses, with a text string to label each port. Clients name their ports by a series of function calls as shown in Figure 8.

```

nameport("view xfm",INPUT0);
nameport("trim xfm",INPUT1);
nameport("obj in",INPUT2);
nameport("obj out",OUTPUT0);

```

Figure 8. Setting port names

The user is presented with a graphical representation of the input and output terminals of each process as its window is pointed to on the screen. Lines indicating connections between terminals are also shown. The terminal labels and connections between them are drawn into overlay planes shared with pop-up menus.

A connection may be made by selecting an output terminal and then selecting an input terminal with the left mouse button. After the connection is made, a line is drawn connecting the output terminal to the input terminal. Connections may be broken by using the middle mouse button to point to terminals of an existing connection.

Connections can be thought of as dependencies. A connection between an output port of one process and an input port of another process makes the input port of the second process dependent on the output port of the first process. **Makecon** and **breakcon** are used to add or delete connections (dependencies).

The connection manager process maintains data structures that duplicate the connection structures in the kernel. This makes it easy for it to draw the connections between all windows on the screen. Also, it can provide appropriate feedback if the user attempts to remove a nonexistent connection. The connection manager is implemented in about 800 lines of C.

8. Related Work

Evans and Sutherland Corporation has a data-flow system for their PS300 graphics system. This environment allows functional blocks to be combined to link input devices to transformations used to display graphical objects. Although it is possible for users to develop function blocks, it is difficult to provide an interactive user interface for a function block. Other problems are that it is not possible to edit the interconnection of processes dynamically, and access to facilities on the host is limited, because the PS300 is a terminal.

Mike Hawley and Sam Leffler have used standard UNIX IPC mechanisms to connect applications in their window system. In [4] they show a font editor created by linking a generic bitmap editor to a program that can display a set of bitmaps. Their system does not support interactive changes to the interconnection of processes.

Early experience with the V-kernel [3] message based IPC created interest in systems capable of supporting arbitrary communication among a group of processes using synchronous messages. The basic V-kernel IPC facilities are quite similar to ours, except the V-kernel does not allow messages to be multi-cast.

We have recently become aware of work Luca Cardelli [1] has done in this area. He has developed a conceptual framework for a system he calls *Fragments of Behavior*. In his system, each fragment has an *interface* for communicating with other fragments, and possibly a *dialog* for communicating with users. The behavior of each fragment is described in Squeak [2], which resembles Hoare's language for communicating sequential processes [5].

The conceptual basis of the work described here closely parallels his in many ways. However, we describe the behavior of our fragments in the C programming language using a single thread of control and receive external events from a single input queue.

9. Future Work

Several things could be done to enhance this applications environment. One thing that is missing is the ability to collect a set of interconnected processes into a single *project* that can be started easily. This type of facility is provided by shell scripts in conventional UNIX. Projects can be used to create pages of dynamic documents [6]. As each page is turned, the state of the current page (project) is saved and the next page (project) is presented.

Many other data-flow utilities are needed to provide convenient access to the file system, record and play back system events, and interface to printers. In a more generalized form, the connection manager process could assist monitoring of data-flow in the system, performing miscellaneous plumbing tasks as needed.

Several things could be done to increase the performance of inter-process communication. Some thought is also being given to replacing the ASCII representation for most objects passed between processes with a general binary *thing-stream* format. This format will contain enough type information so that tools like the object interpolator can operate intelligently.

10. Summary and Discussion

Simple extensions to a conventional operating system have been described that support a data-flow environment. Individual applications react to input events from the user or from other processes. The user interacts with the window manager to direct input focus to a particular data-flow application.

This environment makes it easy to create user interface tools that translate user actions into higher level commands that are sent to another process. This can be used to dynamically bind a user interface to an application.

Interprocess communication is accomplished by typed, variable sized, synchronous messages. Each process has multiple input and output ports, and connection fan-out and fan-in of ports is not limited. Changes in the interconnection of processes may be made without the knowledge of applications programs, by user interaction with a data-flow manager.

Some complexity is added to applications to support interaction with other processes in addition to interaction with the user. Each application is responsible for responding to input events, whether they originate from another process, or from a user. Each process receives input events by reading from an event queue.

It the responsibility of applications to generate new output objects in response to input events. It is usually the goal of each application to keep its outputs consistent with its internal state. This means that as an object is being edited, the editor should write the object to its output port in a timely manner.

The efficiency of interprocess communication could be increased for operating systems where all processes share a single address space. In such an environment, a pointer to an object is all that must be communicated between processes.

Applications of the current system are not limited to graphics. Traditional cutting and pasting of text between windows can easily be implemented within this framework.

One advantage of connections (dependencies) in an interactive environment is the propagation of effects when one process generates output. If a shape editor is connected to an extrusion tool, interactive changes propagate from the shape editor to the extrusion tool, and possibly beyond. This behavior extends the user's view of the *application* to the entire system.

Other dependencies could be used to create a more dynamic environment. The lighting model used by a surface renderer could be made dependent on the room illumination, or the color and intensity balance of the screen could react to the ambient light of the room, or sounds could be generated in response to system events.

We anticipate an end to monolithic *integrated* applications, and prefer *dis*-integration of applications into functional fragments with protocols for communication of objects between fragments.

11. Acknowledgements

We gratefully acknowledge the members of the graphics group at Silicon Graphics for supporting this work. Special thanks go to Peter Broadwell, Tom Davis, Larry Kaplan, Henry Moreton and Rocky Rhodes who contributed many good ideas and developed the IRIS graphics library, and the IRIS window manager.

We would also like to thank David Brown, Jules Bloomenthal, Martin Haeberli, and Pat Hanrahan for providing comments on this paper.

The IRIS workstation deserves recognition for providing enough graphics performance to make this work not only possible, but easy.

References

- [1] Cardelli, L., "Fragments of Behavior", Personal Communication. DEC Systems Research Center, Palo Alto, CA. 1985
- [2] Cardelli, L., and Pike, R., "Squeak: a language for communicating with mice," Computer Graphics 1985.
- [3] Cheriton, D.R., and Zwaenepoel, W., "The distributed V kernel and its performance for diskless workstations," SIGOPS Operating Systems Review (ACM), 17(5) July 1983.
- [4] Hawley, M.J., and Leffler, S.J., "Windows for UNIX at Lucasfilm," Portland USENIX Conference, 1985
- [5] Hoare, C.A.R., "Communicating Sequential Processes," Comm. ACM. 21(8), 1978
- [6] Kay, A. and Goldberg, A. "Personal Dynamic Media", Computer, 10(3), March 1977.
- [7] Rhodes, R., Haeberli, P. and Hickman, K. "Mex - A Window Manager for the IRIS", Proc. Portland USENIX Conference 1985.
- [8] Silicon Graphics Inc., IRIS User's Guide, 1984.
- [9] Smith, Alvy Ray, "Plants, Graftals, and Formal Languages", Computer Graphics, 1984.
- [10] Teitelman, W., "A Display Oriented Programmer's Assistant", Proc. 5th International Joint Conference on Artificial Intelligence, 1977.

Uwm: A User Interface for X Windows

Michael Gancarz

Ultrix Engineering Group
Digital Equipment Corporation
Merrimack, New Hampshire 03054

ABSTRACT

Uwm is a user-programmable user interface client application for X windows. It provides standard functions for moving, resizing, iconifying, raising, and lowering windows, as well as more exotic functions to accommodate specialized window management tasks. These can be bound to mouse buttons and control keys in combinations of your own choosing. *Uwm* also offers pop-up menus of the "slip-off" variety. Menus may contain shell commands, window manager functions, cut buffer strings, and even references to other menus.

Nearly every aspect of *uwm* allows tailoring by the individual user: menu selections and headings, styles of highlighters, font choices, amount of text padding, function contexts, modes of operations, etc., etc. And, if using color hardware, the user has complete control over color choices, a highly subjective area at best. *Uwm* solves the workstation user interface problem in the true UNIX tradition by giving the choices, and ultimately the power, to the user.

Uwm is included in the 4.3 Berkeley distribution along with X windows as user-contributed software.

1. Introduction

X is a network transparent windowing system developed at the Massachusetts Institute of Technology that runs under ULTRIX-32 Version 1.2, 4.2BSD and 4.3BSD UNIX. It provides the familiar model of overlapping windows, as well as many services for the graphics applications programmer. X display servers run on computers with either color or monochrome bitmap terminals. The server multiplexes user input and output requests to and from various client programs located either on the same machine or elsewhere on the network. While a client normally runs on the same machine as the X server it is communicating with, this need not be the case.

A particularly useful feature of X is that, unlike most other windowing systems, the user interface is not part of the server but is a separate client program. Hence, you can choose from a variety of user interfaces, and run them singly or in tandem, on a local or remote machine.

Current user interfaces for X include *xwm*, *xnwm*, and, of course, *uwm*. All provide basic window management functions, such as moving, resizing, stacking and iconifying windows. Each interface is, to a lesser or greater degree, user-

† ULTRIX-32 is a trademark of Digital Equipment Corporation.

‡ UNIX is a trademark of AT&T Bell Laboratories.

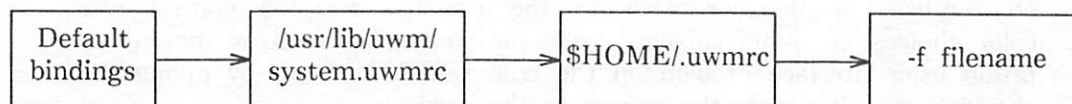
programmable. *Xwm*, the ancestor of both *xnwm* and *uwm*, provides a compact environment with minimum user flexibility and rapid startup. *Xnwm* offers popup menus for window management functions, as well as the ability to tailor mouse button bindings in a dynamic fashion. *Uwm*, too, incorporates popup menus for window management functions, but it carries the popup metaphor one step further by allowing you to specify UNIX commands and "cut buffer" operations in menus. To complement the enhanced menu implementation, *uwm* employs a more sophisticated scheme for specifying mouse button bindings in a variety of contexts.

One design goal of *uwm* was that, for *uwm* to be a truly "user-programmable" interface, it should be capable of emulating nearly every aspect of its X predecessors' functional capabilities. In that respect, it succeeded admirably. It is possible for a user to configure *uwm* to perform identically to *xwm*, and you can create popup menus with *uwm* that incorporate the same functions found in *xnwm*'s popups. Differences remain, but they are largely matters of style, not functionality. However, lest you think that we're playing a game of one-upmanship here, remember that you may still run all of these window managers with the others. None of these interfaces is a "loser" and the user wins in every way.

2. What the User Sees

2.1. The Startup File Search Path

You specify mouse button bindings and menu selections for *uwm* with internal defaults and external bindings found in startup files. The defaults and startup files are recognized in the following order:



Uwm writes the internal default bindings out to a temporary file, parses the file, then unlinks the temporary file. While this might seem inefficient, it insures that the internal defaults have been correctly entered by the *uwm* developer by forcing them to pass through the same error-checking mechanisms that the parser uses for all startup files. The temporary file usually gets no farther than the disk buffer cache before the file is unlinked (except on "write-through" file systems).

System administrators can provide for global default window environments with the file `/usr/lib/uwm/system.uwmrc`. `$HOME/.uwmrc` contains the bindings and menu specifications for an individual user. Using `-f filename`, you can specify an alternate startup file on the command line for use in, say, a programming or text processing environment. If either of `/usr/lib/uwm/system.uwmrc` or `$HOME/.uwmrc` doesn't exist, the program quietly proceeds. A missing *filename* is flagged as an error.

Bindings and menus in the various components of the search path are cumulative. This allows later elements in the chain to take advantage of pre-defined notions of variables, menus and bindings found in earlier files. There are also provisions for overriding anything specified in a previous component.

Uwm uses *yacc(1)* and *lex(1)* to parse the startup files. The parser strips the input of comments, blank lines, and unquoted whitespace, then builds singly-linked lists of menus and mouse button bindings. Illegal mouse binding combinations, references to non-existent menus, and other errors are flagged before the program begins sending commands to the X window server. All errors in the startup file specifications are considered fatal. However, to simplify debugging, *uwm* attempts to report all errors it finds before it exits.

2.2. What's in the Startup File

The startup file may contain variables, mouse button/key binding information, and menu specifications. These may be specified in any order. It isn't necessary to declare a menu before referencing it.

2.2.1. Variables

Uwm supports several variable types. These variables generally have a global effect on *uwm* functions. Boolean, numeric and string types are supported. The variables are covered in detail in the *uwm* manual page, so we'll touch on them briefly here.

The boolean variables, if present, declare that functions will generally have certain characteristics. One example is the *freeze* variable. When *freeze* is set, all "rubber-band boxes" used in resizing or moving windows have a solid appearance; otherwise, the boxes flicker during the operation. Most of the boolean variables can be negated by specifying them as "no<variable>". The *grid* variable, when present, causes an expandable grid to be used instead of a box when resizing. Thus, *grid* can be overridden by specifying *nogrid*.

Numeric variables are used to fine-tune the positions and movements of screen objects, such as windows and icons. Another use for numeric variables is for denoting the amount of "padding" around text used in icons and menus. *Uwm* lets you tailor the placement of text within popup menus and text-style icons. This becomes especially useful when dealing with the more than several dozen text fonts supplied in the typical X implementation.

The most common use of string variables in the startup file is that of specifying fonts. The text fonts used in menus, the text icons supplied by *uwm*, and the font used in a window sizing indicator are all user-programmable. This tends to be a popular feature with users, as font selection is one area where it is impossible to please all the people all the time. *Uwm* responds to this need in a non-dictatorial fashion by saying "Have it your way." For the sake of consistency, however, all menus use the font specified with the *menufont* variable.

2.2.2. Mouse Button/Key Bindings

Mouse button usage under *uwm* is determined by button/key bindings specifications in the startup file search path. It is theoretically possible to tie any built-in window management function to any button combination. *Uwm* owes much of its success (and its criticism) to this feature. You have total control over whether, say, the left mouse button moves windows, the middle button iconifies them, and the right button resizes them. The resultant flexibility, while enormously powerful, has a drawback: The unsuspecting user has no idea what the mouse buttons are used for unless he's using a set of bindings known to him. For this reason, *uwm* is the bane of trade show demonstrators and the joy of experienced users.

The standard format for a mouse/button key binding looks like this:

```
<function> = [<keyspec>] : [<context>] : <button action> [: <menu>]
```

The *keyspec*, *context*, and *menu* fields are optional, but you must include the colon separators regardless. Note that the colon before the *menu* field is required only if the *menu* field exists.

Function is any one of the many window management functions provided. There are functions for moving, resizing, iconifying, and stacking windows in a variety of ways. You can move windows in any direction by fixed increments, relative increments, or by manually positioning them. Windows can be resized by "stretching" one or two sides at a time. Icons normally appear in default positions.

but their positions can be changed during "shrink window to icon" operations by using the *f.newiconify* function. An *f.menu* function makes it easy to bind one or more popup menus to a given button combination. All in all, there are more than a dozen different operations you can choose from, and the number keeps growing as people find more creative things to do with windows.

The *keyspec* field denotes which key(s), if any, on the keyboard must be held down during an entire *uwm* operation. Mouse button presses that occur when the mouse cursor is in an application window could possibly go to the client running in the window or to the window manager. *Uwm* sidesteps the potential conflicts by insisting that when one or two user-specified keyboard "control" keys are held down, all button presses are intercepted by the window manager. The keys are chosen from a group of four possibilities, one of which is the LOCK key. This yields nearly a dozen chording opportunities. If the LOCK key is specified, it makes a handy switch for turning the window manager "off" and "on" while working within an application window. But suppose you aren't using any applications that require the use of the mouse buttons? Fine. A null *keyspec* lets you bypass the use of the keyboard keys at the expense of disallowing the use of the mouse buttons within applications windows.

The *context* field pertains to the position of the mouse cursor when a mouse button action has occurred. The window manager assumes that three possible contexts exist: window, icon and the root (background) window. When the cursor is in a normal window--as opposed to an icon window--when a button is pressed or released, then it is considered to be in a window context. Icon and root contexts apply in a like manner. This capability allows you to specify different actions depending on where the mouse cursor is. For example, when the left button is pressed while the cursor is in a window, you may wish to lower the window in the window stack. But if the left button is pressed while the cursor is in the background, then you may want a full-screen refresh to occur instead. A convenient shorthand makes it easier to include multiple contexts in a binding by combining them with the '|' character. Leaving the *context* field blank means that all contexts are valid for a binding. This is often done when you want a popup menu to appear when a button is pressed, regardless of the mouse cursor position.

Button actions are formed by combining a button (*left*, *middle*, or *right*) with an action of *up*, *down*, or *delta* type. The *down* and *up* actions are intuitive: They occur when a mouse button is pressed or released, respectively. The *delta* action's purpose isn't apparent at first, but it is useful. If the mouse is moved more than *n* pixels in any direction while the button is held down, then the *delta* action takes place. The most common use of this is in binding two different functions to the same button, one to the *up* action and the other to the *delta* action. For example, if the function bound to the *delta* action is *f.iconify* and the function bound to the *up* action is *f.lower*, then the window will be lowered in the window stack if you simply pushed and released the mouse button without moving the mouse. However, if the mouse is moved *n* pixels before releasing the button, then the window will be iconified. Like nearly everything else in *uwm*, *n* is a user-selectable quantity.

Some subtleties exist regarding the relationship of the *context* field and the *button action* field. When you press a button, the current location of the mouse cursor is read by the window manager to determine the relevant context. Upon releasing the button, the location of the cursor is again read to determine the (possibly changed) context. This is done because the function bound to the button *down* action may have altered the context, i.e., the window or icon may no longer be there after the function has taken effect. A still more useful aspect of this is that the user may move the mouse to a different context for the button *up* action. Consider the following bindings:


```
f.iconify = ctrl : window : left down
f.iconify = ctrl : window : left up
```

The *f.iconify* function "iconifies" a window; i.e., it turns a window into an icon. When the left button is pressed with the cursor within a window, that window is iconified. If the user moves the mouse cursor to a different window and then releases the button, the new window will also be iconified. The result is that you have accomplished two functions with one button click. The visual effect on the screen reminds one of a highly adept sleight-of-hand artist.

The thoughtful reader at this point is probably wondering: What context is used for a *delta* action? For a *delta* action to occur, you recall, the mouse cursor must have been moved *n* pixels. In doing so, the context may have changed. After a fair amount of experimentation (and not a little frustration), it was determined that the proper context for the *delta* action is the context when the button was originally pressed. Early *uwm* users found that the position of the mouse cursor when the *delta* action was actually invoked was too unpredictable to use as the point in time to read the context.

The *menu* field in a mouse button binding is required only when using the *f.menu* function. This function paves the way for user-defined menus bound to favorite binding combinations. When the button action bound to the menu function occurs, a popup menu appears on the screen at the mouse cursor position. Which popup menu appears depends on the name in the *menu* field.

You may specify different mouse button bindings with similar menu names, as well as similar mouse button bindings with different menu names. As an example, consider the following bindings:

```
f.menu= meta : root : middle down: "CREATE SMALL WINDOWS"
f.menu= c|s : root : right down : "CREATE SMALL WINDOWS"
f.menu= c|s : root : right down : "CREATE LARGE WINDOWS"
f.menu= c|s : root : right down : "CREATE MANUALLY-SIZED WINDOWS"
```

The first binding shows that pressing the middle mouse button while holding the META key causes the "CREATE SMALL WINDOWS" popup menu to appear. If you select nothing on the menu, the menu disappears and that is that. The last three bindings are an example of *uwm*'s notion of "slip-off" menus. The "CREATE SMALL WINDOWS" popup menu appears when both the CONTROL and SHIFT keys are held down and the right mouse button is pressed. (That may sound confusing, but it's really very simple in practice.) By moving the mouse cursor off the menu without selecting an item, the "CREATE SMALL WINDOWS" menu disappears and the "CREATE LARGE WINDOWS" popup appears in its place. Likewise, the "CREATE MANUALLY-SIZED WINDOWS" popup menu replaces the "CREATE LARGE WINDOWS" menu if no selection is made. The user merely has to brush the mouse to the right to move rapidly from one menu to the next. Hence the name "slip-off" menus. The menus pop into place extremely quickly, even on slow hardware implementations. The reason it works is because, since the user programmed the menus himself, he knows what to expect.

2.2.3. User-programmable Menus

Uwm uses a simple syntax for specifying menus in the startup file. Experience has shown that even unsophisticated UNIX users can create complex menu structures using the commands available. Rather than describe the syntax in detail, let's look at a typical menu:

```

menu "EASY COMMANDS" = {
Move a window:      f.move
Create a window:    !"xterm &"
Edit UWM startup file: !"xterm =65x80+5+5 -e vi $HOME/.uwmrc &"
Access BIGVAX:      !"xterm =24x80+5+5 -e rlogin bigvax &"
Xterm...:           |"xterm -fn 6x10 -i -fg Blue -bg White "
Clear Screen:       ^clear
More -->:           f.menu: "MORE EASY COMMANDS"
}

```

Phrases appearing before the first colon are what the user sees in the popup menu. The portion after the first colon represents the action to be taken when that menu item is selected. The first item shows how the built-in window manager function, *f.move*, can be invoked via a menu to move a window. In fact, all window manager functions may be placed on menus, if desired. The next three lines show how to create various windows via menu selections: One creates a plain window; another runs the *vi* editor on the window manager startup file (notice the shell variable globbing); the third runs an *rlogin* in a window to another host machine. The "Xterm..." item isn't a UNIX command at all. The '|' symbol causes the text following to be placed in the X window server's "cut buffer". The cut buffer's contents can be retrieved at any time and dropped into a terminal emulator window as if the string had been typed on the keyboard. "Clear Screen" is a special instance of a cut buffer operation that appends a newline to the string as it places it in the cut buffer. When the buffer contents are dropped in a window, the UNIX command *clear(1)* is immediately executed. Finally, the "More -->" entry, when selected, erases the current popup menu on the screen and replaces it with yet another menu of commands.

There is tremendous flexibility here. The user can define menus containing items that are most useful for his or her particular operating environment. Window manager functions, UNIX shell commands, text strings, and references to other menus can be freely mixed in any order you wish. Several window managers may be run simultaneously, allowing myriad possibilities--without any knowledge of programming whatsoever.

3. User Interfaces for X-perts

Finding the right interface for every possible user is a formidable task indeed. X windows deals with this issue by not dealing with it. It assumes that an applications developer will write his own user interface and run it as a separate client that exists outside the window server itself. However, most users have neither the time nor the desire to create such a client. Those who have a strict user interface definition will undoubtedly take the time to do so. The rest will probably use *uwm* until something better comes along.

Uwm embodies the same user interface philosophy that X windows does, only at a higher level, i.e., you can't please all the people all the time. But if you let all the people at least have a say in what pleases them, you're well on your way to providing the ideal. *Uwm* gives the user plenty of opportunities for choices. This makes it easy to find an interface that works for him. It may not work well for the next guy, but he'll be quite content with his own environment *because he created it*.

Most of the complaints we've received about *uwm* to date are of the "Why can't I do this or that?" variety. Usually, you can do this or that and much more. But it takes a certain amount of experience with the program to realize its potential. In the interest of speeding up that realization, Appendix A is a copy of the

author's own startup file, edited for brevity. It contains numerous examples of mouse button/key bindings and menus used daily in a software engineering environment. Some of the menus in the example also use color specifications for individual menu items. See the X documentation for details on using color menus in *uwm*, as adequate coverage would require more space than we have here.

The button/key bindings are largely self-explanatory, given a little study. You can guess at what the functions do, as their actions are not so important as the way they are bound. Again, notice that binding different functions to button down and button up lets you combine multiple capabilities on a single button. By pressing a second button before releasing the first, you can abort the button up operation. Also, the ordering of the *f.menu* bindings is significant. That is the order in which they will appear in the "slip-off" sequence.

Smart folks bind the most often used window management functions to button/key combinations and relegate the lesser-used functions to menu items. That way you can invoke them quickly. *Uwm* is perhaps the fastest window manager on today's workstations because of that simple capability.

Most of the menu specifications are straightforward, although some menu selections tend to be a bit creative. One instance is that of the "STDFILE" notion. The STDFILE menu contains an entry that stashes a text string, usually a filename, in a file called *STDFILE*. Other entries in the STDFILE menu can be used to operate on that text string. The possibilities for creating a dynamic command invocation mechanism here are obvious.

The "HANDY COMMANDS" menu contains a potpourri of commands that are usually too painful to type manually. It contains a call to a "SHUTDOWN: Are you sure?" menu which queries the user before running shutdown to prevent accidents. A calendar window is available by running a "browser" program on *cal(1)*. The browser program is really a shell script that *exec*'s its arguments, then waits for a RETURN to be typed on the keyboard. The " $=67 \times 38 + 2 + 2$ " modifier places the calendar at the upper left hand corner of the screen.

Xset(1) is a program that changes various characteristics of the screen interface, such as the background color, keyboard click volume, mouse acceleration, and so on. The numerous references to *xset* show how easy it is to change those characteristics via menu items.

There is much more that can be done in startup files that hasn't even been explored yet. For example, it should be possible to create menu entries that would replace the current startup file with a different one, then restart *uwm* with the new set of bindings. The ability to do shell "globbing" with a menu command bears some experimentation, as do self-modifying startup files. When you combine normal UNIX text processing tools with startup files, the potential increases by an order of magnitude, to say the least. Appendix B contains a shell script that uses *awk(1)* and *ctags(1)* to generate a startup file containing menus which contain names of all functions in a directory full of C source files. Selecting a menu item causes an editor window to be opened with the cursor sitting on the C function in the correct file.

4. Things We'd Like to See Dept.

While *uwm* does a fair amount of work as distributed, there are some features that time did not allow us to put in. Those with source code may feel inclined to make changes. *Uwm* was designed with that goal in mind. Following are some suggestions for those with the urge to hack.

Double Click Action

The mouse button *up*, *down*, and *delta* actions supported suffice for most cases. But just as programmers benefit from more RAM, *uwm* users would like yet another button action. *Xlib(3)* doesn't currently support the kind of lookahead mechanisms needed to do this well. The window manager would have to maintain its own event queue.

Bitmap Menu Items

Text menu items work adequately in most environments. But once in a while it is desirable to add a graphic or two to a menu item for effect.

Dynamic Button/Key Bindings

It would be useful to be able to modify the bindings on the fly and change the run-time environment dynamically. The possibilities for confusing the user would be limitless. Some sort of visual feedback would help.

Alternate Menu Styles

While the built-in "slip-off" menus have proven to be very efficient, why not add a modifier to the menus which would denote an alternate style of display? Pull-downs, dropdowns, and horizontal popups should be fairly easy to implement. Some people have also asked for the slip-off menus to be "reversible". Currently, if you accidentally fall off a menu, you must restart the sequence to return to the original menu. A minor annoyance, but a fix is justifiable.

5. Acknowledgements

The author is deeply grateful to the following individuals and organizations: Paul Asente, DECWRL & Stanford University, for ideas contributed indirectly through *xnum(1)*; Tony Della Fera, MIT Project Athena & DEC, for ideas borrowed from *xum(1)*; Jim Gettys, MIT Project Athena & DEC, for general suggestions and testing; Rich Hyde, DEC Ultrix Engineering Group, for parser contributions; Bob Scheifler, MIT Laboratory for Computer Science, for numerous bug reports; members of the Ultrix Base Workstations Group at DEC Merrimack for their day-to-day testing and support; and Jesus Christ, Heaven, for answering my prayers on how to pull it all together.

6. References

James D. Foley, Andries Van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley Publishing Company, Reading, Mass., 1982

Michael J. Hawley, Samuel J. Leffler, *Windows for UNIX at Lucasfilm*, Usenix Proceedings, Summer, 1985

Jim Gettys, Ron Newman, Tony Della Fera, *Xlib - C Language X Interface*, MIT Project Athena, 1986

Bob Scheifler, *X manual page*, MIT-LCS, 1985

APPENDIX A - Sample Startup File

```
#
# GLOBAL VARIABLES
#
resetvariables;resetbindings;resetmenus
#reverse
autoselect
delta=25
freeze
grid
normali
nonnormalw
hiconpad=5
hmenupad=6
iconfont=oldengssx
menufont=timrom12b
push=1
pushabsolute
resizefont=helv12b
viconpad=5
vmenupad=3
volume=4
#zap
maxcolors=0

#
# BUTTON-KEY BINDINGS
#
# FUNCTION      KEYS CONTEXT  MOUSE BUTTON ACTIONS
f.newiconify=    meta  : window|icon : left delta
f.lower=         meta  : window|icon : left up
f.iconify=       meta  : window      : middle delta
f.iconify=       meta  : icon         : middle down
f.iconify=       meta  : window|icon : middle up
f.moveopaque=    meta  : window|icon : right down
f.raise=         meta  : window|icon : right up
f.circledown=    meta  : root         : left down
f.circleup=      meta  : root         : right down
f.newiconify=    lock  : window|icon : left delta
f.lower=         lock  : window|icon : left up
f.iconify=       lock  : window|icon : middle down
f.move=          lock  : window|icon : right down
f.circleup=      lock  : root         : left down
f.beep =         lock  : root         : middle down
f.circledown=    lock  : root         : right down
f.menu=          m|s   :              : left down   : "WINDOW OPS"
f.menu=          m|s   :              : left down   : " MISCELLANEOUS "
f.menu=          m|s   :              : left down   : " XTERMS"
f.pushleft=      m|s   : window|icon : middle down
f.pushright=     m|s   : window|icon : right down
f.pushup=        m|c   : window|icon : middle down
f.pushdown=      m|c   : window|icon : right down
```



```

#
# MENU SPECIFICATIONS
#
menu = "WINDOW OPS" (White:Black:Black:White) {
"(De)Iconify":(White:"#2f0f2f"):      f.iconify
Move:(White:"#4f0f4f"):                f.move
Resize:(White:"#6f0f6f"):              f.resize
Lower:(White:"#8f0f8f"):               f.lower
Raise:(White:"#af0faf"):               f.raise
Solid Move:(White:"#cf0fcf"):          f.moveopaque
"Others      -->":(White:"#cf0fcf"): f.menu:"EXTENDED WINDOW OPS"
}

menu = "EXTENDED WINDOW OPS" (White:Black:Black:Turquoise) {
Iconify at New Position:(White:"#ff0000"):      f.newiconify
Focus Keyboard on Window:(White:"#d00000"):     f.focus
Freeze Server:(White:"#b00000"):                f.pause
UnFreeze Server:(White:"#900000"):              f.continue
Circulate Windows Up:(White:"#700000"):         f.circleup
Circulate Windows Down:(White:"#500000"):       f.circledown
Refresh Entire Screen:(White:"#300000"):        f.refresh
}

menu = " XTERMS" (White:Black:Red:White) {
*Local*: (White:Black): !"xterm =80x24+0+0 -n local&"
Antic:(White:Blue): !"xterm =80x24+0+0 -s -e antic&"
Decvax:(White:Black): !"xterm =80x24+0+0 -s -e decvax&"
"Unscaled      -->":(White:"#222222"): f.menu:"UNSIZED XTERMS"
"Full-sized    -->":(White:"#666666"): f.menu:"LONG XTERMS"
}

menu = "LONG XTERMS" {
*Local*: !"xterm =80x65+0+0 -n Local -bw 5 -bd Black -cr Black&"
Antic: !"xterm =80x65+0+0 -s -bw 5 -bd Blue -cr Blue -e antic&"
Decvax: !"xterm =80x65+0+0 -n Decvax -s -e decvax&"
}

menu = "UNSIZED XTERMS" {
*Local*: !"xterm -n local -bw 5 -bd Black -cr Black&"
Antic: !"xterm -s -bw 5 -bd Blue -cr Blue -e antic&"
Decvax: !"xterm -s -bw 5 -bd Black -cr Black -e decvax&"
}

menu = " MISCELLANEOUS " (White:Black:Black:White){
"Cut Buffer Strings -->":(White:SkyBlue): f.menu:"CUT BUFFER STRINGS"
"Handy Commands      -->":(White:CornflowerBlue): f.menu:"HANDY COMMANDS"
"STDFILE Menu        -->":(White:SkyBlue): f.menu:"'STDFILE' MENU"
"User Preferences    -->":(White:CornflowerBlue): f.menu:"PREFERENCES"
}

menu = "'STDFILE' MENU" (White:Black:Yellow:Black) {
Edit STDFILE: !"xterm -i =80x65+5+5 -e vi 'cat $HOME/STDFILE'&"
Set & Edit STDFILE: !"xterm -i =80x65+5+5 -e esetstdfile&"
Set STDFILE: !"xterm -i =40x3+100+100 -e setstdfile&"
}

```

```
}
```

```
menu = "CUT BUFFER STRINGS" (White:Black:White:Red) {  
  "Make > tmp &":(Black:White):    ^"rm -f tmp:make >tmp&"  
  "tail -f tmp":(Black:White):      ^"clear;tail -f tmp"  
  "Xted":(Black:White):              ^"xtd -i &"  
  "biff n":(Black:White):            ^"biff n"  
  "xterm -fn 9x15 ...":(Black:White): |"xterm -fn 9x15 "  
  "Clear Screen":(Black:White):      ^clear  
}
```

```
menu = "HANDY COMMANDS" (White:Black:Black:White) {  
  Edit .uwmrc: !"xterm -i =80x65+5+5 -e vi $HOME/.uwmrc&"  
  Restart Window Manager: f.restart  
  Exit Window Manager: f.exit  
  Vi Editor: !"xterm -i =80x65+5+5 -e vi&"  
  Mail Window: !"xterm =+0+0 -e rsh antic mail&"  
  Restart X Server: f.menu:"RESTART SERVER: Are you sure?"  
  Shutdown:f.menu: "SHUTDOWN: Are you sure?"  
  Xtd Editor: !"xtd -i &"  
  1986 Calendar: !"xterm =67x38+2+2 -n 1986 -e browse cal 1986&"  
}
```

```
menu = "SHUTDOWN: Are you sure?" (White:Black:White:Red) {  
  No:(Black:White):    !""  
  Yes:(Black:White):   !"sudo /etc/shutdown -h now"  
}
```

```
menu = "RESTART SERVER: Are you sure?" (White:Black:White:Red) {  
  No:(Black:White):    !""  
  Yes:(Black:White):   !"$HOME/bin/restartX&"  
}
```

```
menu = "PREFERENCES" (White:Black:White:Blue) {  
  Bell Loud:    !"xset b 7&"  
  Bell Normal:  !"xset b 3&"  
  Bell Off:     !"xset b off&"  
  Mouse Fast:   !"xset m 4 2&"  
  Mouse Normal: !"xset m 2 5&"  
  Mouse Slow:   !"xset m 1 1&"  
}
```

APPENDIX B

```
#
# func.mm -- "menu maker" script for 'uwm' window manager
#           Uses ctags to create a series of menus for a uwm startup
#           file. Selecting an item off the menu invokes 'vi -ta'
#           on the appropriate function. The slip-off menus are bound
#           to 'ctrl|shift' on the left button in any context.
#
#           To change the bindings, alter the print statements in the
#           awk portion.
#

FILE=functions.uwmrc
MENUSIZE=10
if test -s $FILE ; then uwm -f $FILE
else
ctags *.c
cat >awktmp <<!
BEGIN {
print "resetvariables:resetbindings:resetmenus:autoselect"
print "delta=25;freeze:grid:hiconpad=5:hmenupad=6"
print "iconfont=oldeng;menufont=timrom12b;resizefont=helv12b"
print "viconpad=5:vmenupad=3:volume=7:zap"
print "f.menu=c|s::left down:\\"EDIT FUNCTIONS\\"
print "menu=\\"EDIT FUNCTIONS\\"(White:Black:White:Red){"
i = 2
}
{ if (NR > $MENUSIZE) {
print ""
printf "f.menu=c|s::left down:\\"EDIT FUNCTIONS #d\\\\"n", i
printf "menu=\\"EDIT FUNCTIONS #d\\"(White:Black:White:Red){\\n", i
NR = 1
++i
}
}
}
{printf "%s:(Black:White):!\\"xterm =80x65+0+0 -e vi -ta %s&\\"n",\ $1,\ $1}
END {print ""}
!
awk -f awktmp tags >>functions.uwmrc
rm -f awktmp &
uwm -f functions.uwmrc
fi
```

Programming with Windows on the Major Workstations. or Through a Glass Darkly

Stephen Daniel
C. Durward Rogers

Microelectronics Center of North Carolina

1. Introduction

In this paper we describe our experiences with porting a quarter of a million lines of interactive VLSI CAD software originally developed on a VAX to the Sun 2/160C and 3/160C, VAX station II, Apollo 550 and 660, Silicon Graphics Iris 2400, Masscomp 500, and the Metheus Lambda 750. We will discuss the general porting problems but concentrate on the difficulties encountered interfacing to the different window systems.

2. What is VIVID

Our painful experience was garnered porting the VIVID system. VIVID is an interactive VLSI-CAD package written entirely in C. It consists of eleven programs and nine subroutine libraries. The programs include an interactive graphics editor for VLSI layouts, a floating-point, computation-intensive circuit simulator, and a compactor which uses neither floating-point nor graphics, but consumes as much virtual memory as is available and manipulates elaborate data-structures.

The graphical editor is fairly old, its interaction model was based on an eight bit-plane color graphics device, a data tablet directly connected to the host computer, and a 24x80 crt. It expects to be able to use all 256 colors and is generally unable to redraw the screen on demand. The circuit simulator has been carefully tuned to have good convergence properties, and is sensitive to the floating point representation. Most of the compactor is easy to port, however it uses small dynamically loaded C routines to configure at run time the procedural translation of symbols, such as transistors, into the polygons that describe their physical structure. While we did not encounter all possible porting difficulties it certainly felt as if we did.

3. The first port - Metheus

For historical reasons, the first workstation we attempted was a Metheus Lambda 750. The Lambda 750 is based on three 68000 processors; one to handle programs, one to handle page faults, and one to run the windowing system. It runs 4.1C BSD Unix and uses its own proprietary windowing system.

Luckily, this was perhaps the easiest window system to interface to. Metheus' window system adopted a strategy of providing lots of hardware support for their windows. A typical Metheus configuration contains sixteen or more bit-planes of graphics memory, and each window on the screen has two or more planes allocated to it when it is created. By manipulating the color map, the window manager is able to make windows appear, disappear, push, and pop with ease. The applications program is never called upon to redraw any portion of the display.

Since this was our first port from a VAX to a 68000 based machine, most of what we learned from the port is that null pointer references tend to creep in whenever you turn

your back on a productive programming team. We solved this problem by implementing a link-time option on our VAX which makes references to page zero illegal.

The only major change we had to make to accommodate their window system was to abandon the separate crt used for alphanumeric display. We did this by having the editor use two windows, one with eight planes for graphics and one with two planes to implement the crt. The editor then had to decide which window should be visible. The Metheus window manager made this system easy to implement. In the Metheus window environment, the alpha window can be opened as a device and data can be written to it at any time using a simple 'write' system call. The window manager provides a good alphanumeric crt emulation, controlled by a standard UNIX termcap entry. This allowed all of the terminal control software in the editor to be used intact.

The Metheus' simple system of providing separate bit-planes for each window is a good hardware intensive solution that makes using the Metheus straightforward for both the programmer and the user. However, in our case, lack of bit-planes precluded running more than one graphical program at a time.

4. The second port - Masscomp

The Masscomp was our first encounter with a System V hybrid. Porting to System V provided no difficulties except for controlling tty lines and signal handling. Our strategy was to isolate all signal handling and terminal control into interface modules, and to have separate versions of these modules for System V and Berkeley systems.

The Masscomp graphics hardware has ten bit-planes of memory — just enough for VIVID to use eight planes for graphics and two planes for a text window. Again, Masscomp supplied adequate software to emulate a terminal on the two text planes, so we were able to port our Metheus version without much pain.

While we were porting VIVID, Masscomp was developing a general window system; however, window systems have long development times, are rarely ready on schedule, and are difficult to figure out without documentation. Their window system should be available and well documented now.

5. Silicon Graphics

Silicon Graphics was our second excursion to a System V hybrid which was, again, surprisingly easy to adapt to. However, this was the only system on which we had significant problems with the floating point implementation. For various reasons (which have to do with making their box run fast), their C compiler implements doubles as 32 bit floating-point numbers. This left us with quite a mess. If we wanted 64 bit numbers to insure the stability of our simulator, we would have to change all "doubles" to "long floats." On the other hand, their graphics system expects doubles, not long floats, so we would have to find all the variables passed to the graphics package and leave them as doubles. If this were not enough, floating point constants default to doubles (which take up less stack space than long floats); therefore, it is not possible to pass a constant to a function expecting a long float without a type cast. Unfortunately, all of these difficulties meant we could not automatically translate our programs into ones that would correctly use 64 bit floating point numbers. In the end, we decided to let the simulator have what trouble it may and run with 32 bit floating point numbers.

In case you are wondering, Kernighan and Ritchie explicitly state that "double" is equivalent to "long float" (p. 193) and that "double" should mean double precision (p. 34). In any case, our experience indicates that the truly portable programmer would always use the term

“long float” in preference to “double.”

The Silicon Graphics window system works but is primitive. It basically provides no support for running more than one graphics program at a time. No information is given to the graphics program when windows might need to be redrawn, and the color map belongs to whichever program last used it. Indeed, if a program comes along and sets all colors to black and exits, you will have trouble using the machine. Also, if you log out without explicitly exiting the window manager, any random person can come up to the system, ignore the login prompt, create a window, and begin using your account.

Silicon Graphics lack of naming conventions in system libraries also caused us trouble. For example, their graphics system has a routine named “select” which conflicts with the name of a system call on Berkeley Unix. Silicon Graphics is not alone in their lack of naming conventions. Of all the workstations we used, only the Apollos used a consistent naming scheme. (Note to implementors of workstation software: come on guys, these are supposed to be state-of-the-art computing tools. Just because the implementors of Unix blew it big time fifteen years ago is no excuse for you not to use naming conventions in your system libraries.)

6. The first hard port - Apollo

At this point in our saga, the marketing department said “market share demands that we support the Apollo workstation” and, of course, we could not argue with this.

As of source release nine (SR9), Apollo supports a fairly comprehensive Unix emulation. We had a little trouble with their C compiler (and we would have had more had we used their 68020 systems), but most of our problems were with their display manager (window system). These seemed to stem from the fact that the Apollo window system is old and originally designed for black-and-white systems.

There were many problems with the window system, all interrelated. The Apollo color systems have eight planes of visible memory. Thus, when VIVID needed to put up an alphanumeric screen, it would obliterate some or all of its graphics screen. Since VIVID is unable to redraw its window on demand, our first thought was to save two of the graphics planes in main memory and use them for the alpha-numeric display. That strategy did not work because the Apollo display manager completely erases all underlying windows when it makes a window invisible. It was not possible to save all of the on-screen memory in main memory and bring it back because this process was unacceptably slow (about ten seconds).

Our second thought was to use the much faster off-screen graphics memory. Unfortunately, the display manager only gives you a tiny fraction of the available off-screen memory and attempts to circumvent it overwrites the current font store and renders further text rather difficult to read. During this process we discovered that, while the Apollo doesn't manage color maps in any useful way, it does prevent the user from changing ten of the first sixteen colors. VIVID of course has its own ideas about what all 256 colors should be.

At that point, we gave up. We wrote our own primitive window system to handle the alpha-numeric screen. It is integrated with our graphics package, and provides a simple tty emulation on the graphics screen. As a result, from the window system's point of view, VIVID now uses only one window. We could not find a way to work around the fact that the Apollo grabs 10 of the first 16 colors, so we performed an elaborate permutation of our standard color map to make use of the Apollo's fixed colors. This eliminated most of our problems with the Apollo.

7. X-Windows on the microVAX II with QDSS board

During the Apollo port, we did a quickie demonstration port to X-Windows running on a prototype VAX station II. As best we could tell from our documentation, each process could ask for however many colors it needed out of the 256 supported by the hardware. These colors then become reserved and unavailable to other processes. Since the window manager itself reserves some colors, we were only able to find about 224 for VIVID to use, with the exact number available depending on the number of windows open. VIVID needs almost all of the 256 colors, so we had to give up. We have since been informed by reliable sources that X-Windows does a much better job than this of allocating color-map resources, and that the problem was in the documentation that was available to us. We had hoped to have a further report ready before the conference but as yet have not received any updated documentation.

8. Suns 2 and 3

Our most recent port is to the Sun 2/160C. This system uses the opposite approach from the Metheus. It is all software with no hardware support for the window system. Its management of the color-map is quite clever: it uses the color-map that corresponds to the window underneath the cursor. Since all VIVID programs use the same color-map, they are quite happy with this system. The Sun package solves VIVID's inability to redisplay the screen by providing a nice system for simultaneously building a main-memory image of the window as it is being built on the screen. At any time, this main memory image can be quickly copied back onto the screen. The concept and implementation were very workable and the only troubles we had were caused by minor bugs in Sun's software. We were able to work around all of them.

9. Conclusions, Dos, and Don'ts

- (1) Get the phone number of someone you can talk to when the window system does not appear to work as advertised.
- (2) Make sure your program can redraw any portion of its graphics window at any time.
- (3) Be prepared to work with a limited number of colors. If possible, be flexible about how many you use and have your program degrade gradually as the number of available colors shrinks. (If you do this well enough, your code will work on a black and white system too.)
- (4) In general, the user of your program will have control over the window size. Be prepared to use different fonts with different size windows, as graphics systems rarely provide nice scalable fonts.
- (5) It is helpful but not necessary to be able to re-scale your screen image if the user changes the window size while the program is running.
- (6) Encapsulate all I/O control calls, signal handling, and asynchronous I/O into interface modules which can easily be re-written for various flavors of Unix. In general, it pays to use only the simplest signaling systems and to avoid asynchronous I/O when possible.
- (7) Use a very simple input paradigm. Do not expect to be able to perform rubber-band or drag echo types. Do not assume that the window system will give you access information about when buttons are released or held.

- (8) If you need to intermix keyboard and graphics input, check to make sure the target systems support multiple input device event queues.
- (9) If you offer a product on a range of workstations, beware that binary data files usually cannot be exchanged between different workstations.
- (10) Finally, avoid all nifty looking features that your window system or Unix system may have, they are sure not to be portable. If you absolutely have to use them, encapsulate as best you can, be prepared to have to implement them yourself on some other workstation, and remember the porter's motto: "I have not yet begun to re-write."

The Influence of Workload on Load Balancing Strategies

Luis-Felipe Cabrera

Computer Science Department
Office System Laboratory
IBM Almaden Research Center ¹
650 Harry Road
San Jose, California 95120-6099
cabrera@ibm.com.ARPA

Abstract

We present an empirical analysis of process lifetimes in several UnixTM installations. We have found consistent processor resource consumption patterns across installations. Moreover, there are process creation patterns present in all the workloads. This analysis shows that several modelling hypotheses used in the load balancing literature do not hold in these installations. We also conclude that for these workloads several proposed load balancing strategies would be detrimental to the user perceived response time. We assert that load balancing strategies in uncontrolled Unix workload environments, such as research and development installations, need to be based on different schedulers than those currently available. We conclude that general purpose load balancing strategies should be based on a process migration mechanism and driven by the detection of long-lived processes. We also conclude that load balancing techniques based on specific services can be implemented at the command level with success.

The tools we have used are available to any Unix user without need of modifying the system.

Section 1. Introduction

All load balancing schemes use the observation that given a set of interconnected processors and a process stream to be executed, there are situations where distributing the processes among processors will be of benefit. Load balancing benefits are sought both in loosely coupled systems (internetworks of computers) as well as in tightly coupled systems (shared memory multiprocessors). The benefit function is usually stated in terms of response time reduction or throughput maxi-

¹ This work was done while the author was with the Computer Science Division of the Electrical Engineering and Computer Sciences Department of the University of California at Berkeley. All the systems measured are in the Computer Science Division of the University of California at Berkeley.

zation. The appeal of load balancing schemes in communities of computers is clear: it is to the best of the community's interest to fully exploit their computer resources.

Research in load balancing strategies has been approached from the modelling and the practical viewpoints by several authors (Barak, Shiloh; Bershad; Bokhari; Bryant, Finkel; Chow, Kohler; Eager, Lazowska, Zahorjan; Krueger, Finkel; Livny, Melman; Stone; Tantawi, Towsley; Wang, Morris). In all modelling approaches the interrelationships between local scheduling cost, distributed load balancing information maintenance cost, and network access and transport cost, are normally oversimplified. Some of the simplifications are required to make the modelling effort mathematically tractable. However, even though it is known that some of these costs, like the network related ones, are non-negligible and non-constant (Cabrera, Karels, Mosher; Hunter), hypotheses which don't take them into account repeatedly surface in the modelling efforts. Workload hypotheses which we have found not to hold, like assuming that all processes belong to a same resource consumption class, are also often found. The overhead associated with the periodic distribution of workload data is seldom taken into account.

The empirical approaches to load balancing we know about have been of the specific command type. In this mode of operation a given system service identified by a command, like text formatting or compiling, is endowed with a front end which determines where to route the request for "better" service. These schemes have been implemented for single services at a time, and not as part of a more general load balancing scheme.

In this paper we report on an initial study of process characteristics in research and development workloads. We have obtained process lifetime data from several installations on different days, as well as process survival rates and process creation patterns. The data collected have characteristics which conflict with several assumptions made about possible load balancing strategies. They also affect their possible benefits. In particular we have seen that most Unix processes are very short lived, thus making very unattractive load balancing mechanisms which operate at the command interpreter level examining all incoming user commands. We have also quantified the fact that "long lived processes tend to live long". We have concluded that schedulers best for load balancing need to identify long lived processes in a system. Current BSD schedulers (Straathof, Thareja, Agrawala) fail to do this.

The rest of the paper is divided as follows. In Section 2 we discuss our data gathering method. Section 3 analyzes process lifetimes and process creation patterns. Section 4 has process survival rates while Section 5 discusses the relevance of our findings to alternative load balancing strategies.

Section 2. Gathering Data

Given that we were interested in the statistical characteristics of processes's resource consumption rather than in very precise measurements about any individual process, we decided to use standard BSD utilities in spite of their inaccuracy. The current sampling method used by BSD accounting facilities is biased as a consequence of the software clock losing interrupts. These clock interrupts are usually set to occur every 10 milliseconds. CPU consumption accounting is also inaccurate as whole time slices are charged to the process which happens to be executing when a clock interrupt

occurs, irrespective of its initiation time within the quantum time slice. Short lived processes, which we have found to be predominant, are the most affected.

From the files updated by the administration utilities of the system we obtained the desired resource consumption statistics. We have concentrated on processor consumption as this is most indicative of possible gains from load balancing. If a process takes little CPU time to complete there is not much improvement in terms of response time and throughput which may come from shipping it to a remote site for execution. Understanding the CPU consumption behavior of processes seems a prerequisite to the adoption of a load balancing strategy.

Our data is of two types. On the one hand we monitored three multi-user systems for a substantial number of days to see, on a day by day basis, the number of processes created in each of these systems. This has given us an indication of how much work such a system can be expected to have. We have also seen how frequent bursts of process creations can be, as we have data showing the number of processes created per 20 minute interval. This data was gathered using a user-level *script*. Our second kind of data is about process birth-time and CPU consumption during its lifetime. This came out of the standard accounting files. We chose five different installations and obtained data about them in different sessions. We repeated our analysis on different days to assess if the data was consistent or not. (It was.) Most of our data gathering sessions were quite long. They typically lasted more than 16 consecutive hours. We also did two short sessions to see if those results matched the others. (They did.) Section 3 and Section 4 have our data and analysis.

For this paper we have labelled "Run 1" up to "Run 10" our data gathering sessions. The five installations considered represent 3 different kinds of processors: VAX 11/750, VAX 11/785, and VAX 8600. Table 1 has the name of the run, the type of processor, an installation identity, and the number of data points in each of the data gathering sessions. All the data depicted in the Figures of this paper comes from VAX 11/785 processors. The installation identities allow comparisons of different data gathering sessions from a given installation.

Label	Installation	Processor	Number of Processes Created
Run 1	c	VAX 11/785	22,732
Run 2	b	VAX 11/785	16,988
Run 3	c	VAX 11/785	21,168
Run 4	c	VAX 11/785	20,490
Run 5	b	VAX 11/785	14,485
Run 6	a	VAX 11/785	26,258
Run 7	e	VAX 8600	1,975
Run 8	d	VAX 11/750	374
Run 9	b	VAX 11/785	11,683
Run 10	a	VAX 11/785	20,471
Total number of processes:			156,624

Table 1: Classification of the data used in this study. Identity of the runs, installations, processors, and number of process creations in each measurement session.

Section 3. Process Lifetimes and Process Creation Patterns

CPU consumption and process creation patterns need to be well understood for load balancing efforts. We shall deal with them in the following two subsections. Indeed process behavior should also be considered when designing or refining the process handling mechanisms of a system.

Section 3.1. Process CPU consumption

Let's define a process lifetime to be the amount, in units of time, of processor cycles it needs to complete.

In BSD environments an approximation of process lifetime is the sum of system time and user time as given by the *time* command. Our measurements show that most processes have a very short lifetime. Indeed faster processors will make this assertion valid for larger fractions of processes. Thus, network latency costs, those incurred when placing an outgoing message on the network transport medium, will have an increasingly important impact in balancing processor loads in an internetwork of fast processors. Say l is the network latency, and t the network transport time between two nodes. Then the (conservative) minimum overhead cost of remote execution is $2l + 2t$, as this is a lower bound on a message round trip time. Given a faster CPU, then, all those processes whose lifetimes are reduced in $2l + 2t$ will automatically not benefit from remote execution.

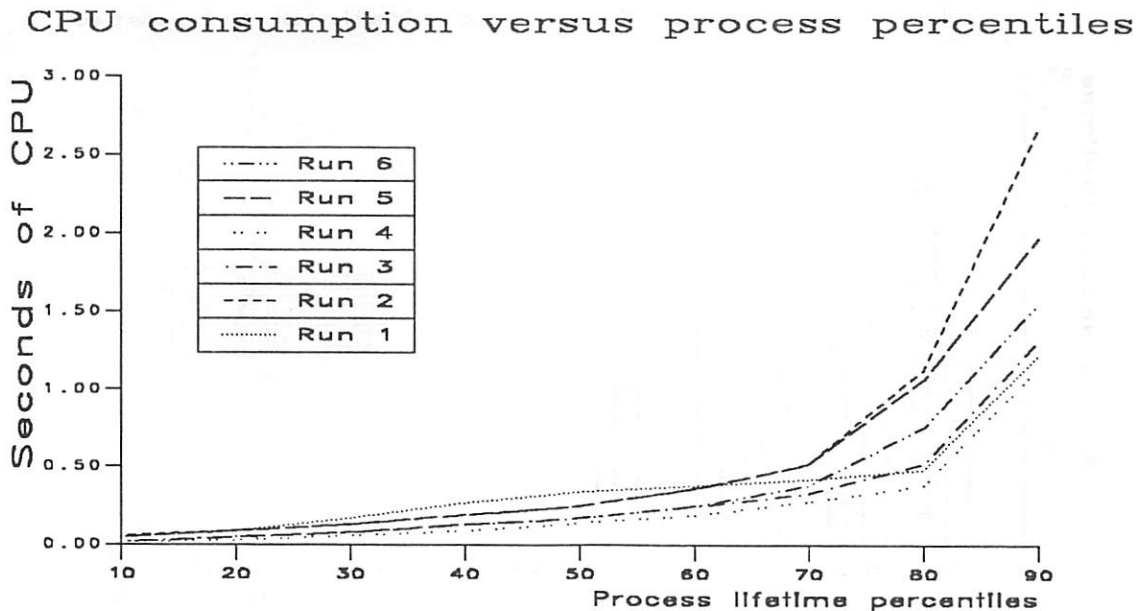


Figure 1: CPU consumption versus process lifetime percentiles.

The high skewness we have observed in process lifetime distributions tells us that the percentage of processes which won't benefit from remote execution increases more than linearly with a linear increase of the CPU power.

Figure 1 shows how the vast majority of processes requires very little CPU. Figure 3 supplements it by displaying longer process lifetimes. The data plotted in these two figures corresponds to more than 122,000 processes, all run on VAX 11/785. Between 70% and 80% of all processes in all the measured installations required less than 0.5 seconds of CPU to execute. (Between 78% and 95% of all processes required less than 1.0 seconds.) As an aside, we can observe from the data that independent of load balancing considerations BSD systems should be tailored for handling short lived processes. This is not the case today. Current process creation and process scheduling mechanisms are not geared for such short lived processes.

Figure 1 also shows that the median lifetime of processes is small. Indeed not only the median lifetime is small but the distributions of process lifetimes are also very skewed. In all systems at least 78% of the processes had lifetimes of less than one second. Figure 2 shows a closer look of processes taking less than one second of CPU, depicting the breakdown in tenths of a second of all processes from three of the runs. Each run was from a different installation. We notice that between 24% and 42% of the 1 second lifetime processes use less that 0.1 seconds of CPU. 73% to 84% require less than 0.4 seconds of CPU. A process based computing environment promotes this behavior.

For short lived processes the soundest processing strategy is to execute them locally. Minimum system overhead should be used when dealing with them. We believe that given today's computing

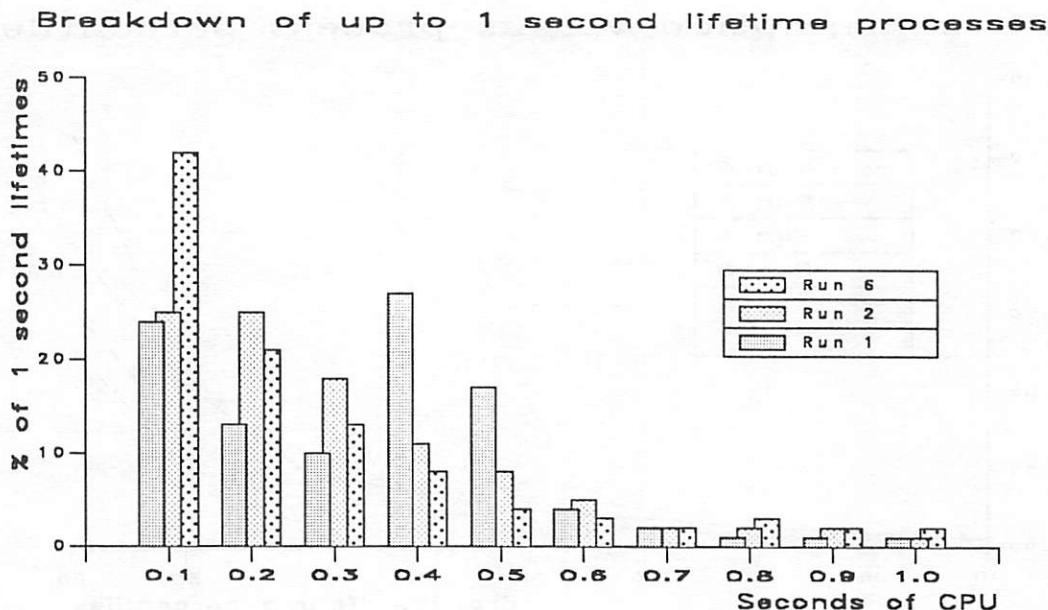


Figure 2: Breakdown of up to 1 second lifetime processes in three different VAX 11/785 installations.

power available even in workstations all load balancing strategies should contemplate local processing of short lived computations. This is certainly compatible with minimization of user perceived response time, and a very good heuristic for throughput optimization of an internetwork of hosts. From the observation that there is a substantial fraction of processes with short lifetimes, it is clear that any load balancing strategy built layered on top of the command interpreter which were to analyze all incoming commands would err on the side of unnecessary overhead. The system should not spend any time deciding to process locally short lived processes. Process oriented operating systems should minimize the overhead of process creation, process dispatching, and process rescheduling.

There are services, however, which are known to require nonnegligible amounts of processor time. It is simple to instrument the system to gather data about them. A command driven approach based on such specific services, like text formatting or compilations, would be able to determine a best processing site with a high degree of confidence. We believe this selective view of command driven load balancing can be successful both for throughput maximization as for response time minimization.

A complementary aspect to the above discussion is that of overall process lifetimes. For how long do the longer lived processes stay alive? Figure 3 is an expanded version of Figure 1, in that we begin with lifetimes of 0.5 seconds and include lifetimes of up to 16 seconds. In Figure 3 we see that in all of the installations not more than 4% of the processes had lifetimes of more than 8 seconds, and not more than 2% had lifetimes of more than 16 seconds. In other words 96% of the processes used less than 8 seconds of CPU to complete and 98% used less than 16 seconds.

It should be remarked that even some of these long lived processes may not be amenable to processing on another host. A long and intense editor session, for example, can consume more than 16 seconds of CPU. It is clearly a mistake to ship that session away for processing. The associated network costs to transport data files back and forth will probably outweighs any response time benefit derived from a faster processor in the target installation. In Section 4 we discuss how load balancing techniques should benefit from these long lived processes.

Section 3.2. Process creations

Workloads are known to exhibit cyclic behavior patterns (Cabrera, Rodriguez-Galant). Figure 4 depicts the process creation behavior of one installation on four different days. The graphs correspond to 24 hour periods based on 20 minute samples. Unix connoisseurs should recognize *cron*'s doings behind the sharp peak at the early hour of the day. In this installation most of the activity is done between 9:30AM and 4:30PM. This same cyclic behavior has been observed in different installations. Thus, there is a correlation across machines between their long term busy and idle cycles.

The maximum number of processes we have seen created per day on a heavily used VAX 11/785 was 33,800 and in a VAX 11/750 was 13,870. The maximum number of processes created in a 20 minute period was 1819, in a 10 minute period was 1189, and in a 5 minute period was 670.

CPU consumption versus process percentiles

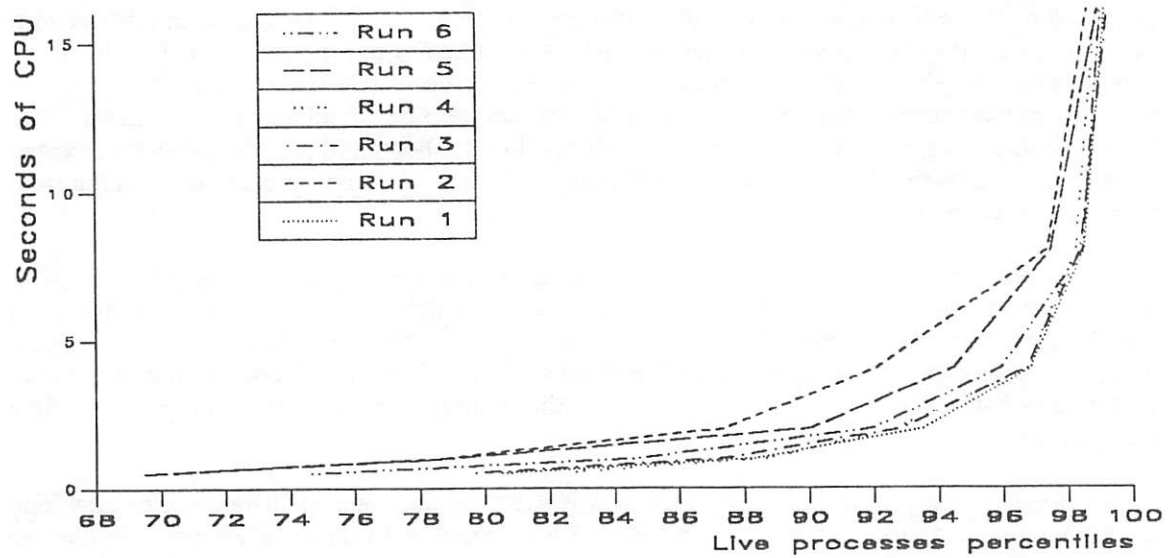


Figure 3: Lifetime percentiles versus total CPU utilization.

Process creations

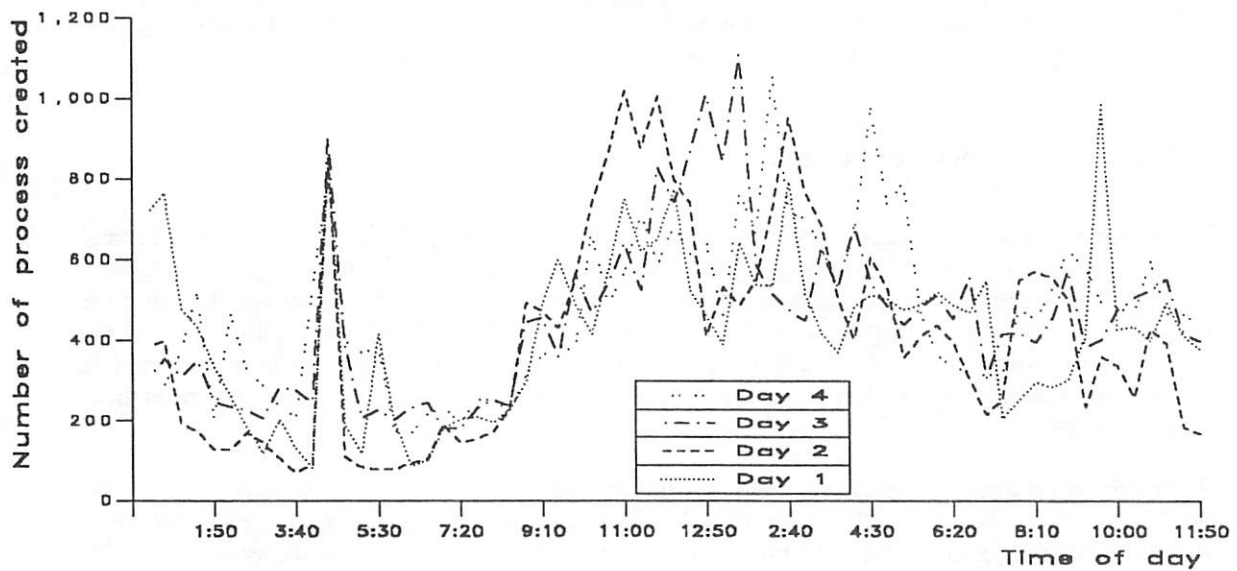


Figure 4: Process creations in 20 minute intervals in four different days. Installation a.

Figures 5 and 6 shed light on process creation frequencies on Run 9 and 10. Figure 5 depicts the frequency of periods in which different amounts of processes were created in one second windows. In Run 9 we see that in 94% of the one second intervals there was at most one process created. This figure was 86% for Run 10. In Section 3.1 we have seen the substantial fraction of processes that requires less than 0.5 seconds of CPU time to complete. Thus, for this time granularity we see that load balancing techniques based exclusively on process arrival considerations should probably do nothing between 86% to 94% of the time. An added consideration for such load balancing strategies is the high cost they would pay to keep "live" the workload data in one second intervals. Experience with high level BSD utilities, such as *rwhod*, have shown this overhead to be substantial.

Figure 6 depicts the frequency of periods in which different amounts of processes were created in five second windows. We now see that for Run 9 in 60% of the five second intervals there was at most one process created. This figure was 40% for Run 10. A ratio of elapsed time to CPU utilization time of 10 is high. Figures 5 and 6 tell us that all of those processes which use at most 0.5 seconds of CPU time should be processed locally.

The possible coincidence across systems of bursts of process creations, hinted by Figure 4 albeit for an inappropriate granularity, supports the statement that load balancing techniques for internetworks of computers with uncontrolled workloads should only consider long lived processes.

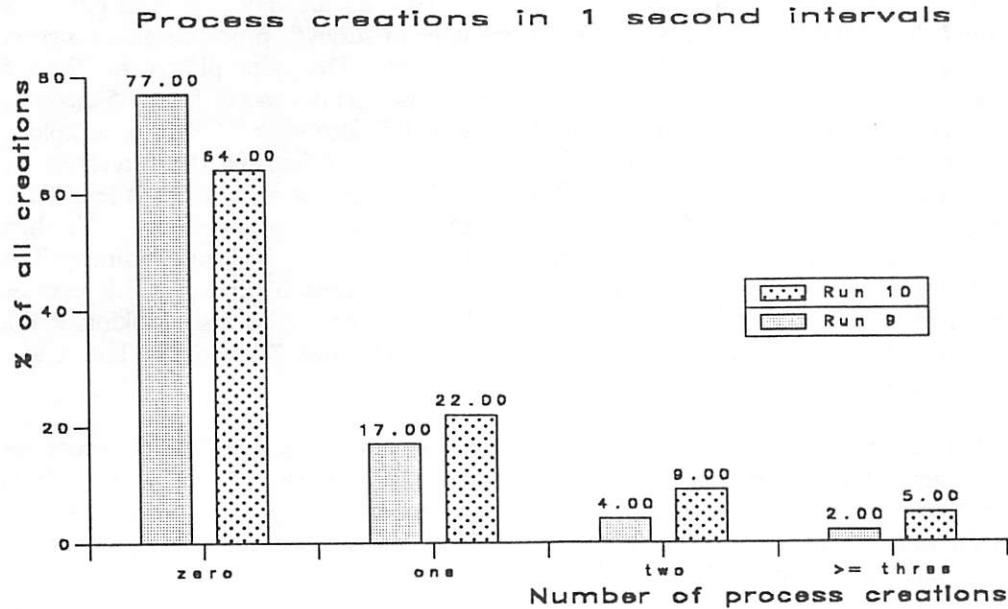


Figure 5: Process creations on Run 9 and Run 10 in 1 second intervals.

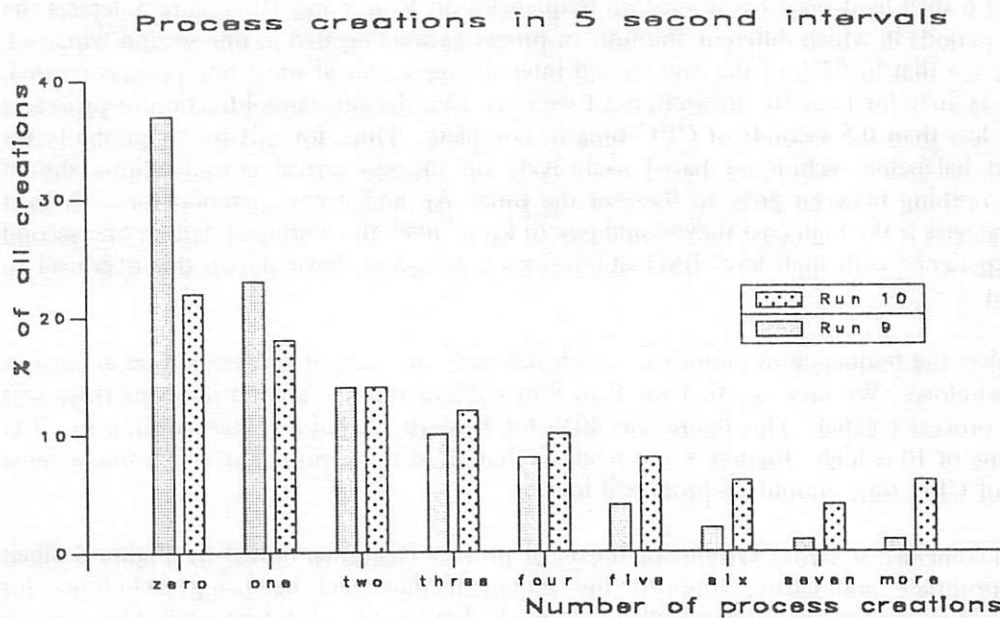


Figure 6: Process creations on Run 9 and Run 10 in 5 second intervals.

Section 4. Process Survival Rates

We have seen that most processes have very short lifetimes. How do the very long lived processes behave? Figure 7 shows this. We have plotted the percentage of survivor processes as a function of process lifetimes in the range of 0.5 seconds to 1024 seconds. The value plotted for the 0.5 second lifetime corresponds to the percentage of processes whose lifetime was at least 0.5 seconds, i. e., the percentage of processes created who took at least 0.5 seconds of CPU to complete. Beginning with processes whose lifetime was 0.5 seconds, we then calculated the percentage of them whose lifetime was larger than 1.0 seconds. The plots show that for all systems at least 60% of those processes whose lifetime was 0.5 seconds had a lifetime of at least 1.0 second. We then looked at the percentage of processes with 1.0 second lifetime who had 2.0 second lifetimes. The plots show that for all systems at least 44% of those processes whose lifetime was 1.0 seconds had a lifetime of at least 2.0 seconds. The rest of the plots proceed analogously looking at the percentages of processes with lifetimes of T seconds whose lifetimes was $2T$ seconds. The X axis in Figure 7 has a logarithmic scale.

The remarkable result is that for a wide range of lifetimes and systems at least 40% of the processes which have a lifetime of T units of time have in fact a lifetime of $2T$ units of time. This confirms that general purpose load balancing techniques should be concerned only about very long lived processes, relying on process migration for their redistribution. To achieve this other parts of the system, like the scheduler, should collaborate by detecting the long-lived processes.

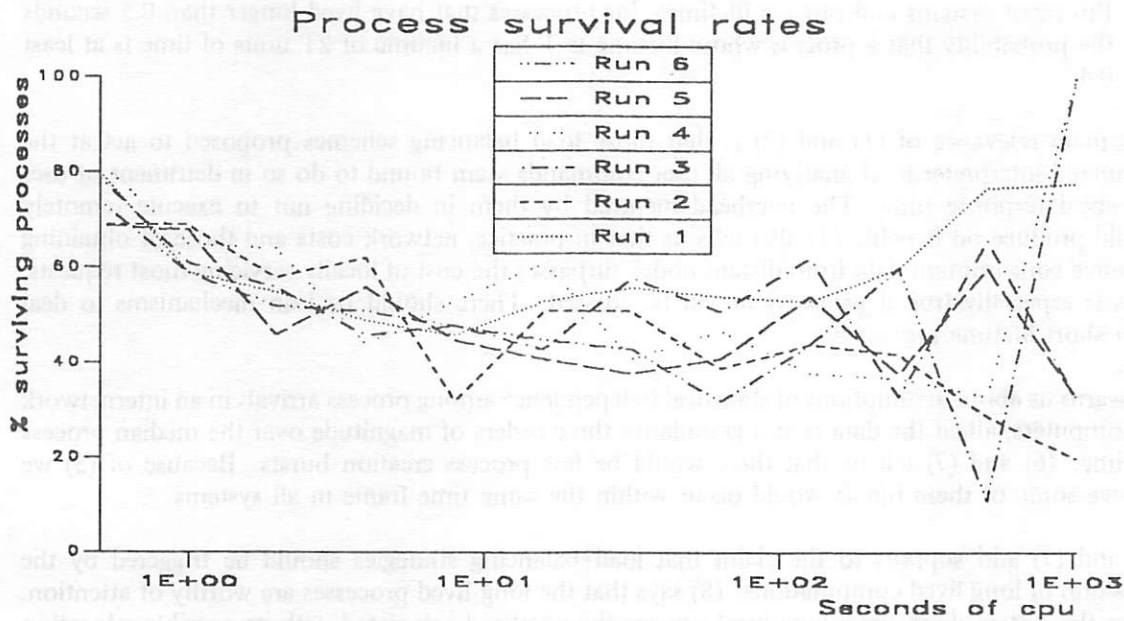


Figure 7: Process survival rates versus total CPU utilization. For each lifetime beginning at 0.5 seconds, the value of Y represents the fraction of processes whose lifetimes exceeded twice the previous lifetime.

Section 5. Conclusions on Load Balancing Strategies

Our previous sections have shown the following assertions for the measured BSD systems:

1. The median lifetime of a process is 0.4 seconds.
2. More than 78% of processes have lifetimes of less than 1 second.
3. 97% of processes have a lifetime of less than 8 seconds.
4. 98% of processes have a lifetime of less than 16 seconds.
5. Installations exhibit cyclic process creation patterns.
6. There are installations where at most one process is created in 94% of all one second intervals of wall clock time.
7. There are installations where at most one process is created in 60% of all five second intervals of wall clock time.

8. For most systems and process lifetimes, for processes that have lived longer than 0.5 seconds the probability that a process whose lifetime is T has a lifetime of $2T$ units of time is at least 0.4.

The main relevance of (1) and (2) is that those load balancing schemes proposed to act at the command interpreter level analyzing all user commands seem bound to do so in detriment of user perceived response time. The overhead incurred by them in deciding not to execute remotely would produce no benefit. (1) also tells us that in practice, network costs and those of obtaining resource consumption data from distant nodes surpasses the cost of locally servicing most requests. This is especially true if gateways are to be crossed. There should be lean mechanisms to deal with short lifetime processes.

(5) warns us about assumptions of statistical independence among process arrivals in an internetwork of computers, albeit the data is at a granularity three orders of magnitude over the median process lifetime. (6) and (7) tell us that there would be few process creation bursts. Because of (5) we believe some of these bursts would occur within the same time frame in all systems.

(6) and (7) add support to the claim that load balancing strategies should be triggered by the detection of long lived computations. (8) says that the long-lived processes are worthy of attention. Once the system discovers a long-lived process the overhead associated with its possible relocation is well spent.

Looking at the scheduler of most Unix installations, we see that there is a pervasive use of Round-Robin with priority queues. There is not, however, a scheduler understood notion of long-lived processes. This would allow, for example, to suspend and reschedule at a future time in possibly a different machine a long-lived process. In high load conditions when several long-lived processes are competing among themselves for processor cycles the fairness criterion built into the current Unix schedulers prevent suspension of such processes. Schedulers will have to change to make good use of the knowledge that a particular process is long-lived (specially if it is a noninteractive one). A categorization along these lines would allow triggering a load sharing mechanism which could query other systems about their long term resource demands. These long term resource demands can be better predicted if one has an explicit category of long-term jobs.

In light of these measurements, one should also consider possible changes to the mechanisms which manipulate processes. The basic observation is to minimize process creation costs. One can also think of making the system data statistics optional. They should be allowed to be turned off at will. Lastly process dispatching and process scheduling priority (re)computations should have minimum overhead.

In synthesis, the empirical analysis of process lifetimes shows that the workload has an important impact on load balancing strategies. Our data clearly shows that in the measured environments a command interpreter based general purpose load balancing scheme which were to analyze all incoming user commands would be detrimental to both response time and throughput. We have also seen that the scheduler will need to play an important role in successful load balancing strategies. The observation that between 78% and 95% of all processes have a lifetime of less than 1.0 seconds points us to the long lived processes as those worthy of attention for load balancing. The fact that long-lived processes survive longer also calls for this long-lived process based approach. Our data suggests that the only viable general purpose load balancing schemes

should be based on a process migration mechanism and driven by the detection of long-lived processes. Moreover, there should also be a mechanism to differentiate long lived interactive processes from batch oriented intense computations. The former should not be migrated at all while the latter are the prime candidate for load balancing. We also conclude that load balancing techniques based on specific services can be implemented at the command level with success.

We conjecture that other workloads of systems which favor the process model of computing will also exhibit the behaviors described in this paper. For such workloads all of our considerations apply.

Acknowledgements. Thanks to Ricardo Gusella for providing me with some data. Also thanks to Roger Haskin, John Palmer, and Jim Wyllie for their helpful comments and careful reading of earlier versions of this manuscript.

Section 6. References

1. A. Barak, and A. Shiloh, A Distributed Load Balancing Policy for a Multicomputer, Department of Computer Science, The Hebrew University of Jerusalem, 1984.
2. B. Bershad, The Garcon-Maitre d' System. Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley.
3. R. Bryant, and R. A. Finkel, A Stable Distributed Scheduling Algorithm, Proceedings of the 2nd International Conference on Distributed Computing Systems, April 1981.
4. L.-F. Cabrera, and G. Rodriguez-Galant, Predicting Performance in Unix Systems From Portable Workload Estimators Based on the Terminal Probe Method, Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Research Report UCB/CSD 84/194, August 1984.
5. L.-F. Cabrera, E. Hunter, M. Karels, and D. Mosher, A User-Process Oriented Performance Study of Ethernet Networking Under Berkeley Unix 4.2BSD. Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Research Report UCB/CSD 84/217, December 1984.
6. L.-F. Cabrera, M. Karels, and D. Mosher, The Impact of Buffer Management on Networking Software Performance in Berkeley Unix 4.2BSD: A Case Study. Proceedings of the 1985 Summer Usenix Conference, Portland, Oregon, pp. 507-517. June 1985.
7. D. L. Eager, E. D. Lazowska, and J. Zahorjan, Adaptive Load Sharing in Homogeneous Distributed Systems. IEEE Transactions on Software Engineering, December 1985.
8. D. L. Eager, E. D. Lazowska, and J. Zahorjan, A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing. To appear in Performance Evaluation, Spring 1986.
9. E. Hunter, A Performance Study of the Ethernet Under Berkeley Unix 4.2BSD. Proceedings of CMG XV, pp. 373-382, December 1984.

10. P. Krueger, and R. A. Finkel, An Adaptive Load Balancing Algorithm for a Multicomputer. Technical Report 539, Department of Computer Science, University of Wisconsin, April 1984.
11. M. Livny, and M. Melman, Load Balancing in Homogeneous Broadcast Distributed Systems. Proceedings of ACM Computer Network Performance Symposium, 1982.
12. H. S. Stone, Multiprocessor Scheduling with the Aid of Network Flow Algorithms. IEEE Transactions on Software Engineering, January 1977.
13. H. S. Stone, Critical Load Factors in Two Processors Distributed Systems. IEEE Transactions on Software Engineering, May 1978.
14. J. H. Straathof, A. K. Thareja, and A. K. Agrawala, Unix Scheduling for Large Systems, Proceedings of the 1986 Winter Usenix Technical Conference, Denver, Colorado, pp. 111-139.
15. A. N. Tantawi, and D. Towsley, Optimal Static Load Balancing in Distributed Computer Systems. JACM 32, No. 2, April 1985.
16. K. Thompson, Unix Implementation, Bell System Technical Journal, Vol. 57, No. 6, Part 2, pp. 1931-1946, July/August 1978.
17. Y.-T. Wang, and R. J. T. Morris, Load Sharing in Distributed Systems. IEEE Transactions on Computers, C-34, No. 3, March 1985.

A System V Compatible Implementation of 4.2BSD Job Control

David C. Lennert

Hewlett-Packard Company
Information Technology Group
hplabs!hpda!davel

ABSTRACT

This paper gives an overview of how process groups and controlling terminals are handled in System V and 4.2BSD and then discusses the effect 4.2BSD job control has on these things. A modified 4.2BSD interface is discussed which supports 4.2BSD job control functionality but in a way which allows AT&T System V compatibility. This interface has been implemented in Hewlett-Packard's UNIX[†] system, HP-UX.

1. INTRODUCTION

The job control functionality first introduced into UNIX by Jim Kulp of IIASA and later provided by 4.2BSD UNIX has become a *de facto* industry standard. However, this job control facility, as implemented in 4.2BSD, is incompatible in several respects with System V.

Recently a proposal was submitted to the IEEE P1003 Portable Operating System standard committee by Sun Microsystems [Harris86] which attempts to define 4.2BSD job control functionality in a way compatible with System V. Hewlett-Packard Company has been independently developing a similar proposal. HP's proposal is almost identical to Sun's but goes beyond it to address many "corner case" areas which strongly affect System V compatibility.

This paper gives an overview of the relevant areas of System V functionality which are affected. It then overviews how job control is implemented in 4.2BSD and how this impacts the System V interface. Finally, the HP-UX interface is presented and a similar overview of its implementation is given.

The various overviews cover how job control signals are generated, passed, and acknowledged by the tty driver and user processes. They also explain how process groups are established and changed.

2. FUNDAMENTALS

In the following discussion the reader is assumed to have an understanding of several fundamental concepts found in the UNIX operating system. For convenience these concepts are briefly reviewed here.

[†] UNIX is a trademark of AT&T.

2.1. Process Groups and Controlling Terminals

Every process has a unique numeric value associated with it called its *process ID*. Every process also has a non-unique numeric value associated with it called its *process group ID*. A *process group* is a collection of processes having identical numeric process group ID's. Typically, one process in the process group will be the *process group leader*. The process group leader has a process ID which is numerically equal to the process group ID associated with all processes in the process group. Typically, the process group leader is the ancestor of all other processes in the process group.

A process can have a *controlling terminal* which is usually the login terminal of the user who created the process. A process can obtain access to its controlling terminal by opening the file `/dev/tty`. All processes in the same process group typically share the same controlling terminal. A terminal usually has a process group ID associated with it, called the *tty group ID*. When a user generates a keyboard signal (e.g., by typing the interrupt character), the tty driver sends the appropriate signal to all processes which are members of the process group indicated by the tty group ID. In summary, usually, but not necessarily, all processes in the same process group share the same controlling terminal, and the tty group ID for that terminal is equal to the process group ID of the process group.

For further explanation see [Roch85] and intro(2) in your favorite UNIX Programmer's Manual.

2.2. 4.2BSD Job Control

4.2BSD job control allows users to selectively stop (suspend) the execution of processes and continue (resume) their execution at any later point. This only easily works for processes which are stopped and continued during the same login session.

The user almost always employs this facility via the interactive interface jointly supplied by the system tty driver and a job control shell such as `csh(1)` or `ksh(1)`. The tty driver recognizes a user-defined *suspend character* which causes all current foreground processes to stop and the user's job control shell to resume. The job control shell provides commands which continue stopped processes in either the foreground or background. The tty driver will also stop a background process when it attempts to read from or write to the users terminal. This allows the user to finish or suspend their foreground task without interruption and continue the stopped background process at a more convenient time.

To enable the system to support this, 4.2BSD job control introduces five new signals: `SIGSTOP`, `SIGTSTP`, `SIGTTIN`, `SIGTTOU`, and `SIGCONT`. The first four signals cause a process to stop unless the signals are being caught or ignored. `SIGCONT` always causes a stopped process to continue. (`SIGCONT` has no effect on processes which are not stopped.) `SIGSTOP` cannot be caught or ignored.

The tty driver sends some of these signals to all processes in the tty process group under the following conditions: The driver sends `SIGTSTP` when the user types the suspend or delayed suspend character. The driver sends `SIGTTIN` (`SIGTTOU`) when a background process attempts to read from (write to) its controlling terminal. `SIGCONT` is usually only sent by a job control shell when the user requests that a stopped process be continued. Of course, any signal can be sent by a user via the `kill(1)` command or by a program via the `kill(2)` system call.

It should be noted that these signals can be added to a UNIX implementation in a manner which preserves source and object code compatibility. A process is not required to be aware of them. By default the signals do "the right thing."

For further information see [Joy80] and [UCB83].

3. AT&T SYSTEM V

3.1. Introduction

System V process groups closely resemble the concept of a login session. That is, all processes spawned during the same login session tend to belong to the same process group, and keyboard signals are typically sent to all processes spawned from the login session.

3.2. System V Process Group Handling

In System V, the only way to alter the process group associated with a process (`p_pgrp`) is via `setpgrp(2)`. And this can only set the process group to equal the process ID (`pid`) of the process. When this happens the resulting process with `pid = p_pgrp` is called a process group leader. Since a process's `pid` can never change, once a process issues a `setpgrp(2)` call it irrevocably becomes a process group leader.

The `init(1M)` process spawns all other processes on the system either directly or indirectly. Before directly spawning a process (after the `fork(2)` but before the `exec(2)`), `init` calls `setpgrp(2)`. Thus all original children (not orphans) of `init` are forced to (irrevocably) be process group leaders.

When a new process is created, it is assigned a new `pid` but it inherits the process group number of its parent. Thus child processes are, by default, not process group leaders (although they can become a process group leader via `setpgrp(2)`).

When a process group leader which has a controlling terminal (see below) terminates, `SIGHUP` is sent to all processes in the same process group.

Further, when a process group leader terminates, all processes that belong to this process group are altered to belong to no process group (their `p_pgrp` is set to zero). More precisely, when any process exits, all processes whose process group (`p_pgrp`) equals the `pid` of the terminating process will have their `p_pgrp` set to zero; this check succeeds only in the case of a terminating process group leader.

3.3. System V Controlling Terminals

A terminal that is currently open by a process may also be a "controlling terminal" for a process group. When certain control characters are typed on a controlling terminal, signals are sent by the terminal driver to all processes that belong to the process group associated with the terminal.

When a process becomes a process group leader (via `setpgrp(2)`) it automatically loses its controlling terminal. After this, the first terminal (that is not already a controlling terminal) opened by the process is assigned to be the controlling terminal for that process. Also, the process group associated with that terminal (`t_pgrp`, also known as the `tty` group ID) is set equal to the process group associated with the process group leader (`p_pgrp`). All child processes inherit the controlling terminal and process group of their parent.

More precisely, in System V, the process group associated with a terminal (`t_pgrp`), can be changed in the following ways:

- (1) When a terminal is opened by a process group leader (`pid == p_pgrp`) that does not already have a controlling terminal, it becomes the controlling terminal for that process group (`t_pgrp` is set equal to `p_pgrp`) if it is not already a controlling terminal.
- (2) When a process group leader (`pid == p_pgrp`) dies, if it has a controlling terminal that is associated with the same process group (`t_pgrp == p_pgrp`), then that terminal is disassociated from that process group (`t_pgrp` is set to zero).
- (3) When the last process to have a terminal open closes that terminal, the terminal is disassociated from its process group (`t_pgrp` is set to zero).

3.4. System V Typical Scenario

This is a typical scenario for the birth and death of a process group and its controlling terminal.

The `init(1M)` process wants to enable a terminal for login. It calls `fork(2)` to create a new process and then calls `setpgrp(2)` to make the process a process group leader which also removes the process's controlling terminal. It then runs the `getty(1M)` program as the process via `exec(2)`. `Getty` opens the terminal causing it to become `getty`'s controlling terminal and be associated with `getty`'s process group (`t_pgrp` is set to `p_pgrp`). `Getty` replaces itself with `login(1)` which replaces itself with a login shell, e.g., `sh(1)`. Usually no program calls `setpgrp(2)` and thus all descendent processes of the login shell are in the same process group and have the same controlling terminal; keyboard signals are sent to all processes launched during this session.

When a logout occurs, the login shell (which is the process group leader) dies and the controlling terminal is freed up (`t_pgrp` is set to zero) so that it can be claimed as a controlling terminal by a subsequent `getty` respawned by `init`. `SIGHUP` is sent to all processes in the same process group. The process group (`p_pgrp`) of all descendent processes is then set to zero.

Note that there may continue to be background processes (previously started by the now defunct login shell) which continue to execute but keyboard signals will no longer be sent to these processes (since both `t_pgrp` and `p_pgrp` equal zero).

4. 4.2BSD

4.1. Introduction

4.2BSD process groups closely resemble the concept of a task within a login session, where a task represents a set of processes which are affected as a group by job control operations. Every time a job control shell (e.g., `csh`) spawns either a foreground or background command, all processes in the pipeline (and their descendants) are placed in their own unique process group with the first command in the pipeline being the process group leader.

A task is in the foreground when the process group associated with the controlling terminal for the task (`t_pgrp`) is equal to the process group associated with the processes in the task (`p_pgrp`). Otherwise the task is in the background. A job control shell moves a job between the foreground and background by adjusting the terminal process group (`t_pgrp`) of the controlling terminal.

Note that 4.2BSD forms new process groups with process group leaders much more often than System V usually does (every command versus every login).

4.2. 4.2BSD Process Group Handling

In 4.2BSD, the process group associated with a process (`p_pgrp`) can be altered in two ways. The first is via `setpgrp(2)`. 4.2BSD's `setpgrp(2)` is analogous to System V's `setpgrp(2)` except that the former can affect processes other than the current process and can cause the affected process to adopt a process group other than that process's process ID (`pid`). Thus, unlike System V, a process can cease to be a process group leader.

In addition to `setpgrp(2)`, a process that is not a member of any process group (`p_pgrp == 0`) will "inherit" or join the process group associated with its controlling terminal at the time the process is assigned a controlling terminal during `open(2)`. If the terminal being opened is not presently the controlling terminal for any process group, then the process opening the terminal will first be made a process group leader (`p_pgrp` will be set to `p_pid`) and then the terminal will become the controlling terminal for this new process group. All this is done by the `tty open` code.

When a new process is created it inherits the process group of its parent.

Unlike System V `init(1M)`, 4.2BSD `init(8)` does not call `setpgrp(2)` when spawning other processes. All processes spawned by `init` inherit `init`'s process group which happens to be zero ("not a member of any process group"). This is actually crucial for assigning controlling terminals; see below.

4.3. 4.2BSD Controlling Terminals

Unlike System V, a 4.2BSD process does not lose its controlling terminal when altering its process group (via `setpgrp(2)`). Also unlike System V, a 4.2BSD process that is a process group leader (`pid == p_pgrp`) but which has no controlling terminal does not receive a controlling terminal when opening a new terminal.

A process can obtain a controlling terminal under 4.2BSD in only the following ways:

- (1) A process can inherit a controlling terminal from its parent.
- (2) A process that is not a member of any process group (`p_pgrp == 0`) can open any terminal and that terminal will become its controlling terminal (whether or not it is already the controlling terminal for another process). However, this can happen in one of two ways:

If the terminal is not already a controlling terminal (`t_pgrp == 0`) then the opening process becomes a process group leader (its `p_pgrp` is set equal to its `pid`) and the terminal becomes its controlling terminal (`t_pgrp` is set to the new `p_pgrp` value).

If the terminal is already a controlling terminal for another process (`t_pgrp` is not zero) then the opening process joins the process group already associated with the controlling terminal. That is, `p_pgrp` is set equal to the current `t_pgrp`. Note that the opening process does not become a process group leader, i.e., `p_pgrp` is not equal to its `pid`.

Note that this procedure only happens during the first terminal open for a process that was either originally spawned by `init` or whose ancestor processes (all the way back to `init`) never altered their process group (`p_pgrp`) either by opening a terminal or calling `setpgrp(2)`.

A terminal ceases to be a controlling terminal (`t_pgrp` is set to zero) under 4.2BSD in the following way:

- (1) When the last process to have a terminal open closes that terminal then the terminal is disassociated from its process group (`t_pgrp` is set to zero).

There are two other facilities unique to 4.2BSD which affect access to control terminals: the `TIOCSPGRP ioctl(2)` and `vhangup(2)`.

The `TIOCSPGRP ioctl(2)` function changes a terminal's process group (`t_pgrp`) to any desired value. It is typically used by `csh(1)` to control which set of processes (process group) is in the foreground.

The `vhangup(2)` function is invoked by `init` after forking but before `exec'ing` `getty`. This function removes read and write permission for all processes (including the caller) that have the controlling terminal open (whether or not it is their controlling terminal). It then sends `SIGHUP` to the process group associated with the terminal (`t_pgrp`). The latter action is similar to the System V functionality that sends `SIGHUP` to a process group on death of the process group leader; 4.2BSD does not do this on the death of a process group leader.

4.4. 4.2BSD Typical Scenario

This is a typical scenario for the birth and death of a login, its controlling terminal, and process groups associated with a job.

The `init(8)` process wants to enable a terminal for login. First it creates a new process via `fork(2)`. Then it opens the terminal which (because the `p_pgrp` inherited from `init` is zero) causes it to become the controlling terminal for this process and either alters the

process group (`p_pgrp`) of the process to match the terminal process group (`t_pgrp`) if non-zero, or alters both `p_pgrp` and `t_pgrp` to equal the process ID (`pid`) if `t_pgrp` is zero. At this point the new process has a controlling terminal whose process group (`t_pgrp`) is equal to the process's process group (`p_pgrp`). However, the process may not be a process group leader (i.e., `p_pgrp` may not equal `pid`).

Now the new process calls `vhangup(2)` to remove access permissions for the controlling terminal from all processes (as well as sending `SIGHUP` to any processes in the process group previously associated with the terminal). The new process then reopens the terminal to get a file descriptor with read and write permissions since the `vhangup(2)` removed these permission from the file descriptor returned by the previous `open`. The previous file descriptor is not closed until now to prevent losing the controlling terminal; (remember that `p_pgrp` for the new process is no longer zero.)

The new process now replaces itself with `getty(8)` which replaces itself with `login(1)` which replaces itself with a login shell, e.g., `csh(1)`. `Csh` now begins to manipulate the process group associated with the terminal (`t_pgrp`) via the `TIOCSPGRP` and `TIOCGPGRP` `ioctl(2)` calls and the process group associated with its child processes (`p_pgrp`) via `setpgrp(2)` in order to allow job control. This happens (briefly) in the following way:

`Csh` launches a pipeline by making all programs in the pipeline be immediate descendants of `csh`. (This is different from `sh` which makes all programs in the pipeline except the last be descendants of the last program in the pipeline.) All programs in the pipeline belong to the same process group (not the same as `csh`'s process group) and the first program in the pipeline is the process group leader (its `pid` is equal to the process group for the pipeline). If the pipeline is being launched in the foreground (or moved to the foreground) then the process group associated with the terminal (`t_pgrp`) is set to the process group of the pipeline.

When a logout occurs, the login shell dies. Any pending `SIGTTIN`, `SIGTTOU`, and `SIGTSTP` signals are cleared for all descendent processes. All immediate child processes are inherited as orphans by `init`; if any are currently stopped then they are killed (`SIGKILL`). If the exiting process is the last process that has the controlling terminal open then the terminal's process group (`t_pgrp`) is set to zero, otherwise it is left alone. Nothing special is done for process group leaders; in fact, login shells are frequently not process group leaders. (`SIGHUP` is not sent and the controlling terminal is not necessarily cleared.)

Note that there may continue to be processes (previously started by the now defunct login shell) which continue to execute. And that keyboard signals can still be sent to these processes under some circumstances (specifically when the processes were in the foreground (`p_pgrp == t_pgrp`) when the login shell died; this usually only happens when the login shell is killed from another terminal via `kill(1)`.) Note also that this continues to be true even after a new session logs in on the same terminal since the new login shell joins the process group which is already associated with the terminal from the prior login.

4.5. Job Control Signal Handling

The following discussions concerning signals and kernel process synchronization are similar to ones found in [Thom78], [Ritch79], and [Bach79].

4.5.1. Basic Overview

Usually a process is either running or sleeping waiting for an event to occur (e.g., I/O completion). When a signal is sent to a process (either by another process or an I/O driver) what actually occurs is that a flag is set for the receiving (or target) process indicating that the signal has been sent and the target process performs the actual signal operation to itself the next time it runs. Thus sending a signal amounts to requesting the target process to itself perform a particular action. If the target process is already running it is interrupted to process the signal. If it is runnable but not currently running then the system merely waits for it to become the currently running process at which point the signal is acknowledged. If the

target process is sleeping then either it is moved into a runnable state (if it is sleeping at an "interruptable" priority) or it is left sleeping (at an "uninterruptable" priority) and the signal is not acknowledged until the slept on event occurs.

The kernel procedure which sends a signal is `psignal()` and is executed by the sending process or driver. `Psignal()` updates a list of pending signals for the receiving process. If the receiving process is the currently running process and it is executing in kernel mode then the pending signal is acknowledged when the current system call completes. (This is the case where the sending process and the receiving process are the same.) If the receiving process is the currently running process and it is executing in user mode then a special event is generated which causes the process to enter the kernel and acknowledge the pending signal. (This is the case where the sending "process" is really an interrupt handler which, for example, is servicing an interrupt character typed on a user's terminal.) If the receiving process is sleeping but not holding off signals then it is set running via `wakeup()`; the pending signal is acknowledged as soon as the receiving process executes. If the receiving process is suspended in a sleep state that holds off signals ("sleeping uninterruptably") then it is left sleeping; the pending signal will be acknowledged after the waited for event occurs.

The procedure which tests for a pending signal is `issig()` and is executed by the receiving process. `Issig()` is executed whenever the receiving process changes from kernel mode to user mode execution; for example, at the completion of a system call. It is also executed whenever the receiving process is awakened from being suspended in a sleep state that does not hold off signals ("sleeping interruptably").

The procedure which performs the requested signal operation (e.g., invoking a signal handler or killing the process) is `psig()` and is executed by the receiving process if `issig()` returns true.

This basic structure is essentially the same in System V, 4.2BSD, and HP-UX. However, under 4.2BSD-style job control, these general principles can work slightly differently:

When processing stop signals, the `psignal()` function, called by the sending process, actually stops the target process sometimes. In these cases, the target process never realizes that it received the signal or that it stopped. However, in other cases, `psignal()` performs the usual process of setting the flag (`p_sig`) requesting that the target process stop itself the next time it runs.

The `issig()` function, called by the target process, can actually stop the target process.

The `psig()` function is only called in the case where a user handler has been provided for the job control signal.

A more complete description of job control signal handling is contained in the pseudocode below.

4.5.2. `psignal()`

To send `SIGCONT` to a target process:

```
sending SIGCONT clears any pending stop signals;
if the target process is STOPPED but is also SLEEPING (p_wchan != 0)
    merely continue the process's SLEEP;

else if the target process is STOPPED and is NOT SLEEPING (p_wchan == 0)
    set the process RUNNING;
```

To send a stop signal (`SIGTSTP`, `SIGTTIN`, `SIGTTOU`, `SIGSTOP`) to a target process:

```
sending a stop signal clears any pending SIGCONT;

if the target process is RUNNABLE or RUNNING
```

```

        note the pending signal in p_sig;

    else if the target process is SLEEPING NON-interruptably
        note the pending signal in p_sig;

    else if the target process is SLEEPING interruptably
        and IS catching the signal
        note the pending signal in p_sig and
        wakeup the process from its sleep;

    else if the target process is SLEEPING interruptably
        and is NOT catching the signal
        stop the process by setting its state to SSTOP
        but leave it sleeping on its p_wchan;
        send SIGCLD to parent (if it expects BSD-style)

```

General note: sending a stop signal (other than SIGSTOP) to a child of init causes the target process to be killed.

4.5.3. issig()

Issig() is called in all cases except where the process was sleeping interruptably and was not catching the signal.

To acknowledge a pending SIGCONT or stop signal:

```

    if in the middle of a VFORK
        hold off all stop signals (pretend they don't exist yet)

    else if catching the signal
        return a request to invoke user signal handler via psig()

    else if SIGCONT
        do nothing /* pretend it doesn't exist */

    else /* stop signals */
        stop the process by setting its state to SSTOP
        send SIGCLD to parent (if it expects BSD-style)
        call swtch() to dispatch another process

```

General note: sending a stop signal (other than SIGSTOP) to a child of init causes the target process to be killed.

4.5.4. psig()

Psig() is called whenever issig() returns an indication that a user handler is defined for a job control signal. Psig() merely invokes the user signal handler.

4.5.5. wakeup()

The fact that a process is sleeping (waiting for an event to occur) is indicated by two process state values: p_wchan is non-zero, indicating the event being waited for, and the process state is SSLEEP.

Wakeup() usually causes all processes waiting (sleeping) on a specified event to be awakened. When a process is awakened two things happen: The process is removed from the sleep queue (p_wchan is cleared) and it is added to the run queue.

If, however, `wakeup()` discovers a process whose `p_wchan` matches the specified event but whose process state is `SSTOP` (stopped) then the process is removed from the sleep queue (indicating that the waited for even has happened) but it is not placed on the run queue. A subsequent `SIGCONT` will cause it to be placed on the run queue.

Thus it is possible to have a process which is both sleeping (`p_wchan` non-zero) and stopped (process state is `SSTOP` rather than `SSLEEP`).

4.5.6. Signal Setup via `init`

When `init(8)` launches any process it causes the process to ignore all the job control stop signals (`SIGTSTP`, `SIGTTIN`, & `SIGTTOU`). This allows login shells which are not job control shells to automatically ignore the signals. Further, all descendent processes of such a login shell will also ignore these signals unless they explicitly enable them.

4.6. Foreground/Background Processes

4.6.1. Basic Overview

4.2BSD job control supports the notion of a process being in the *foreground* or *background*. The distinction is a background process is usually forced to stop when it attempts to perform I/O (including most control operations) on its controlling terminal, while a foreground process is not hindered.

Specifically, when a background process attempts to read from its controlling terminal it is sent the `SIGTTIN` signal which, by default, causes it to stop. When it attempts to write to its controlling terminal and `LTOSTOP` has been enabled for the terminal, then the process is sent the `SIGTTOU` signal which, by default, causes it to stop. If, however, a background process has chosen to catch the signal, the specified user handler is invoked. If the process is ignoring or masking the stop signal(s), then the terminal I/O request returns an I/O error, `EIO`.

A background process is one whose process group (`p_pgrp`) is not equal to the process group of its controlling terminal (`t_pgrp`) (and `t_pgrp` is not zero). All other processes (including ones doing I/O to terminals that are not their controlling terminals) are considered to be in the foreground.

4.6.2. Tty Driver Provisions

To distinguish between foreground and background programs the tty driver must perform checks on attempted I/O operations to a process's controlling terminal. This is done in several places.

At the beginning of a read/write system call the tty driver checks to see if the calling process is in the background. If it is, then all processes in the process group of the calling process are sent the appropriate signal (`SIGTTIN` or `SIGTTOU`) unless the signal is masked or ignored by the calling process. In this case the driver returns the `EIO` error. After the tty driver sends the signal, the calling process is put to sleep waiting for the *lightning bolt event*[†]. This allows the calling process to receive the signal (and usually stop). When the process returns from the sleep (usually by being continued) the tty driver repeats the foreground/background check before proceeding with the operation.

When the process is in the foreground, the I/O operation proceeds. In the case of a terminal read this usually results in the process being put to sleep to wait for input characters to arrive. At this point the user could type their suspend character (e.g., `^Z`). This causes the interrupt portion of the tty driver to send `SIGTSTP` to the controlling terminal's process group (i.e., all processes which are in the foreground). In our scenario this would include the

[†] The lightning bolt event is a standard UNIX event which occurs frequently, for example, every second.

process sleeping on terminal input, and this would typically cause it to stop.

When a sleeping process is stopped it is also left sleeping as well. If, in this case, tty input characters subsequently arrived then the process would be awakened. However, because it is also stopped, it would not be set running; it would merely be "unslept".

At some later time the process would be continued (e.g., via a csh "fg" or "bg" command which sends SIGCONT). If the process had not been previously unslept it would merely continue its sleeping; it would receive no indication that it had stopped and continued. If the process had been previously unslept it would now be set running.

When the process is set running it resumes execution in the tty driver. Because a (potentially substantial) amount of time has elapsed and because the process may have been stopped and restarted, the tty driver is no longer sure whether this process is still in the foreground. So before checking if input characters are available, the tty driver rechecks whether the process is in the foreground or background. This is necessary because, in our scenario, the stopped process could have been continued in the background (via csh "bg"). To check this the tty driver merely repeats the foreground/background check it made at the beginning of the system call.

5. SYSTEM V INCOMPATIBILITIES AND THEIR RESOLUTIONS

Job control as implemented in 4.2BSD is incompatible with System V semantics in some significant respects. This section discusses each of these incompatibilities and the resolution implemented in HP-UX to maintain System V compatibility.

The system interface needed to support 4.2BSD-style job control, tailored for System V compatibility as discussed in this section, is presented in the form of manual page excerpts in [Len86].

5.1. Setpgrp(2) Changes

Because the needed semantics of 4.2BSD setpgrp(2) conflict with the semantics of System V setpgrp(2), the 4.2BSD setpgrp(2) function was renamed to be setpgrp2(2). (The choice of new name is arbitrary; setpgrp2 was chosen in the same spirit as 4.2BSD's wait3(2).)

5.2. SIGHUP Changes

System V semantics state that when a process group leader dies, all processes in the same process group are sent the SIGHUP signal which, by default, kills all the processes.

Job control shells execute a command by making all processes in the pipeline belong to the same (brand new) process group and by making the first program in the pipeline be the process group leader. Typically, the first program in a pipeline terminates before the other programs. Under System V semantics, this would cause the premature death of the remaining pipeline. Because of this, 4.2BSD does not generate SIGHUP on process group leader death.

In order to support System V semantics and still allow job control to function properly, HP-UX makes a distinction between a "System V process group leader" and a "job control process group leader". A System V process group leader is given System V semantics (SIGHUP is generated) and a job control process group leader is given 4.2BSD semantics (SIGHUP is not generated). A process which becomes a process group leader via setpgrp(2) is considered to be a System V process group leader. A process which becomes a process group leader via setpgrp2(2) is considered to be a job control process group leader. Since the HP-UX (and System V) init(1M) program calls setpgrp(2) on behalf of all processes it spawns, all login shells start out as System V process group leaders. A process must explicitly call setpgrp2(2) to deviate from the System V semantics.

5.3. SIGCLD Changes

Under System V, SIGCLD is sent to a process whenever one of its immediate child processes dies. Under 4.2BSD, SIGCLD (or its variant, SIGCHLD) is also generated when a process changes state from running to stopped. Since a System V application would not expect to receive SIGCLD under these new circumstances and since a job control shell would not be able to function properly without such notification, a compatible compromise was developed.

The (parent) process wishing to trap SIGCLD may set a flag when calling the HP-UX `sigvector(2)`[†] routine to establish a signal handler. This flag will cause SIGCLD to be sent for stopped children, in addition to terminated children. A System V application using `signal(2)` will see the System V compatible SIGCLD semantics.

5.4. Controlling Terminal Changes

Under System V, whenever a process group leader dies, the controlling terminal associated with that process group (if any) is deallocated (disassociated from that process group). 4.2BSD does not deallocate controlling terminals on process group leader death for the following reason: Job control shells make the lead process in every pipeline a process group leader. If the controlling terminal for each pipeline were deallocated whenever the lead process terminated, then the remaining processes would effectively become background processes (assuming they were currently in the foreground) and would stop when any of them attempted subsequent I/O to the terminal.

To allow both semantics, controlling terminals are only deallocated when a "System V process group leader" dies and not when a "job control process group leader" dies. (See the discussion of SIGHUP changes above.)

However, this change leads to the following problem: In order for a terminal to be allocated as a controlling terminal for a new login, it must be deallocated when the previous login terminates. System V relies on process group leader death to deallocate controlling terminals (since all login shells are forced to be process group leaders by `init(1M)`). This is no longer reliable since login shells could become "job control process group leaders". Further, not all logins are spawned directly by `init(1M)`; the 4.2BSD `rlogin` facility is a prime example. 4.2BSD solves this problem by allowing a new login to join the process group of the controlling terminal which is still allocated from the previous login. However this violates System V compatibility.

The solution chosen was to mark a process that causes a controlling terminal to be allocated and to deallocate the controlling terminal whenever that process terminates. This reliably catches logins which are spawned either directly or indirectly from `init(1M)`, whether they are "System V process group leaders" or not. Controlling terminals continue to be deallocated on death of System V process group leaders using the System V semantics.

5.5. Security

Several security holes exist in the 4.2BSD process group altering mechanisms. To plug these holes the following changes were made.

4.2BSD `setpggrp(2)` allows a process to alter the process group associated with another process to any value. 4.2BSD restricts this operation so that the affected process must pass the same security restrictions enforced when sending signals, or must be a descendent of the calling process. However, this still allows a process to join a process group already associated with another user. To tighten this security, `setpggrp(2)` was further restricted such that if the

[†] `Sigvector(2)` is an HP-UX extension proposed to the IEEE P1003 [Head85] which supports both the reliable signal operations of 4.2BSD `sigvec(2)` and the conventional signal operations of System V `signal(2)`. In HP-UX, `signal(2)` is implemented as a library using `sigvector(2)`. Note that the changes proposed here to `sigvector(2)` can be identically made to 4.2BSD `sigvec(2)`.

specified new process group value is equal to the process ID (pid) or process group ID of any existing processes, then all such processes must pass the above security restrictions.

Similarly, the 4.2BSD TIOCSPGRP ioctl(2) allows a terminal's process group to be altered to any value. This allows a user's terminal to easily become an additional "controlling terminal" for another user's process group; keyboard signals can be sent to the other user's processes, thus bypassing the security enforced by kill(2). Because of this, the TIOCSPGRP ioctl(2) was altered to enforce similar security restrictions as setpgrp(2).

In System V and 4.2BSD, a process can obtain access to its controlling terminal by opening the file `/dev/tty`. Under System V, processes left executing after a user's logout are allowed further access to `/dev/tty` until the terminal it represents is reallocated as a controlling terminal for a new login. More specifically, `/dev/tty` access is allowed whenever the process group ID of the leftover process matches the process group ID of the terminal. These IDs continue to match immediately after logout (since both have been zeroed) until the terminal is re-enabled for login by `getty(1M)`. (Note that when the new login terminates, `/dev/tty` access is restored again to these prior processes because the controlling terminal's process group ID is re-zeroed.) Further, if a process has its controlling terminal opened directly (not via the `/dev/tty` synonym) then access is not restricted at all after logout. These System V semantics can constitute security problems. However, they are not explicitly required by the System V Interface Definition [ATT86].

4.2BSD does nothing to hamper `/dev/tty` access for processes remaining after logout. The process group ID for the controlling terminal is not altered, and, in fact, it is preserved even into the next login (since subsequent logins join the already existing process group associated with the terminal, if any). These semantics also represent security problems. However, 4.2BSD does prohibit access to the controlling terminal if it is opened directly; this is accomplished when `init(8)` issues the `vhangup(2)` system call.

Although preserving the System V semantics for controlling terminal access after logout is not deemed necessary or even recommended, it is easy to do in the following way. Whenever a process that allocated a controlling terminal dies, all processes which share this controlling terminal have their process group ID zeroed. This is analogous to, and occurs in addition to, the System V behavior of zeroing the process group ID for all related processes when their process group leader dies. `/dev/tty` checks similar to System V can then be employed.

5.6. TTY Driver Considerations

For System V compatibility, the suspend and delayed suspend characters are defaulted to a disabled value (0377). This means that job control is "inactive" by default when a user logs on. The user must explicitly activate job control by defining either or both of these characters via `stty(1)` or some similar interface.

There should be no problem allowing 4.2BSD-style job control, as modified here, to co-exist with System V's shell layers job control system. (See `shl(1)` and `sxt(7)` in the System V Release 2 reference manuals.)

6. HP-UX

6.1. Introduction

HP-UX process groups are used in two major ways.

System V process groups closely resemble the concept of a login session. That is, all processes spawned during the same login session tend to belong to the same process group, and keyboard signals are typically sent to all processes spawned from the login session.

Job control process groups closely resemble the concept of a task within a login session, where a task represents a set of processes which are affected as a group by job control

operations. Every time a job control shell (e.g., `csh`) spawns either a foreground or background command, all processes in the pipeline (and their descendents) are placed in their own unique process group with the first command in the pipeline being the process group leader.

A task is in the foreground when the process group associated with the controlling terminal for the task (`t_pgrp`) is equal to the process group associated with the processes in the task (`p_pgrp`). Otherwise the task is in the background. A job control shell moves a job between the foreground and background by adjusting the terminal process group (`t_pgrp`) of the controlling terminal.

Note that a job control shell forms new process groups with process group leaders much more often than a non-job control (System V) shell usually does (every command versus every login).

6.2. HP-UX Process Group Handling

In HP-UX, the process group associated with a process (`p_pgrp`) can be altered via `setpgrp(2)` or `setpgroup(2)`.

As in System V, `setpgrp(2)` can only set the process group to equal the process ID (`pid`) of the process. When this happens, the resulting process with `pid = p_pgrp` is called a System V process group leader.

`Setpgroup(2)` is analogous to `setpgrp(2)` except that it can affect processes other than the current process and can cause the affected process to adopt a process group other than that process's process ID (`pid`). `Setpgroup(2)` also forms job control process groups rather than System V process groups. Using `setpgroup(2)`, the calling process, or certain other processes, can either become a job control process group leader or can cease to be a process group leader.

Because job control process groups are handled slightly differently by HP-UX than System V process groups, HP-UX marks processes that are job control process group leaders (i.e., that have called `setpgroup(2)` without subsequently calling `setpgrp(2)`).

The `init(1M)` process spawns all other processes on the system either directly or indirectly. Before directly spawning a process (after the `fork(2)` but before the `exec(2)`), `init` calls `setpgrp(2)`. Thus all original children (not orphans) of `init` are forced to start out as System V process group leaders.

When a new process is created, it is assigned a new `pid` but it inherits the process group number of its parent. Thus child processes are, by default, not process group leaders (although they can become a process group leader via either `setpgrp(2)` or `setpgroup(2)`).

When a System V process group leader that has a controlling terminal (see below) terminates, `SIGHUP` is sent to all processes in the same process group. Further, when a System V process group leader terminates, all processes which belong to this process group are altered to belong to no process group (their `p_pgrp` is set to zero).

Also, whenever any process that allocated a controlling terminal terminates, all processes that share this controlling terminal are altered to belong to no process group (their `p_pgrp` is set to zero).

When any process exits, any pending `SIGTTIN`, `SIGTTOU`, and `SIGTSTP` signals are cleared from all descendent processes (not just immediate children).

6.3. HP-UX Controlling Terminals

A terminal that is currently open by a process may also be a "controlling terminal" for a process group (collection of processes). When certain control characters are typed on a controlling terminal, signals are sent by the terminal driver to all processes which belong to the process group associated with the terminal. These include the job control suspend and delayed suspend characters.

Controlling terminals also play a role in determining whether a process is in the foreground or background. See FOREGROUND/BACKGROUND PROCESSES above.

When a process becomes a System V process group leader (via `setpgrp(2)`) it automatically loses its controlling terminal. (This does not happen for a job control process group leader, i.e. when calling `setpgrp2(2)`.) After this, the first terminal (that is not already a controlling terminal) opened by a process that is a (System V or job control) process group leader is assigned to be the controlling terminal for that process; also the process group associated with that terminal (`t_pgrp`) is set equal to the process group associated with the process group leader process (`p_pgrp`). All child processes inherit the controlling terminal and process group of their parent.

More precisely, in HP-UX, the process group associated with a terminal (`t_pgrp`), can be changed in the following ways:

- (1) When a terminal is opened by a System V or job control process group leader (`pid == p_pgrp`) that does not already have a controlling terminal, it becomes the controlling terminal for that process group (`t_pgrp` is set equal to `p_pgrp`) if it is not already a controlling terminal.
- (2) When a System V process group leader (`pid == p_pgrp`) dies, if it has a controlling terminal that is associated with the same process group (`t_pgrp == p_pgrp`), that terminal is disassociated from that process group (`t_pgrp` is set to zero).
- (3) When any process dies which originally caused a controlling terminal to be created (see (1) above), if it still has a controlling terminal, that terminal is disassociated from its process group (`t_pgrp` is set to zero).
- (4) When the last process to have a terminal open closes that terminal, the terminal is disassociated from its process group (`t_pgrp` is set to zero).
- (5) The `TIOCSGRP ioctl(2)` call can explicitly change a terminal's process group (`t_pgrp`) to any value within certain security restrictions. This is used by a job control shell to change which set of processes (process group) is in the foreground.

6.4. HP-UX Typical Scenario

This is a typical scenario for the birth and death of a login, its controlling terminal, and the process groups associated with a job.

The `init(1M)` process wants to enable a terminal for login. It creates a new process via `fork(2)` and calls `setpgrp(2)` to make it a System V process group leader which also removes its controlling terminal. `Init` then runs the `getty(1M)` program as the process via `exec(2)`. `Getty` opens the terminal causing the terminal to become `getty`'s controlling terminal and be associated with `getty`'s process group (`t_pgrp` is set to `p_pgrp`). As a side effect, this process is now marked as having created a controlling terminal; when it dies the controlling terminal will be freed for re-use. `Getty` replaces itself with `login(1)` which replaces itself with a login shell.

At this point one of two scenarios typically takes place. The login shell is either a job control shell (e.g., `csh(1)`) or it is not (e.g., `sh(1)`).

If the login shell is not a job control shell then things proceed much as they do on System V. Usually no program calls `setpgrp(2)` or `setpgrp2(2)` and thus all descendent processes of the login shell are in the same process group and have the same controlling terminal; keyboard signals are sent to all processes launched during this session.

If the login shell is a job control shell, then job control operations are performed. `Csh` begins to manipulate the process group associated with the terminal (`t_pgrp`) via the `TIOCSGRP` and `TIOCGGRP ioctl(2)` calls and the process group associated with its child processes (`p_pgrp`) via `setpgrp2(2)` in order to allow job control. This happens (briefly) in the following way:

Csh launches a pipeline by making all programs in the pipeline be immediate descendants of csh. (This is different from sh which makes all programs in the pipeline except the last be descendants of the last program in the pipeline.) All programs in the pipeline belong to the same process group (not the same as csh's process group) and the first program in the pipeline is the process group leader (its pid is equal to the process group for the pipeline). This process is specially marked as a job control process group leader since it was established via setpgrp(2); this basically prevents SIGHUP from being sent to the pipeline when the lead process dies. If the pipeline is being launched in the foreground (or moved to the foreground) then the process group associated with the terminal (t_pgrp) is set to the process group of the pipeline via the TIOCSPGRP ioctl(2).

When a logout occurs, the login shell dies. Any pending SIGTTIN, SIGTTOU, and SIGTSTP signals are cleared for all descendent processes. All immediate child processes are inherited as orphans by init; if any are currently stopped then they are killed (SIGKILL). Since the login shell (actually the getty before it was overlaid) created a controlling terminal, the controlling terminal is now freed (t_pgrp is set to zero) so that it can be claimed as a controlling terminal by a subsequent getty respawned by init; also, all processes which share this controlling terminal have their process group (p_pgrp) set to zero.

When a logout occurs and the login shell is a System V process group leader, SIGHUP is sent to all processes in the same process group, and the process group (p_pgrp) of all descendent processes is set to zero.

Note that there may continue to be background processes (previously started by the now defunct login shell) which continue to execute but keyboard signals will no longer be sent to these processes (since both t_pgrp and p_pgrp equal zero).

7. ACKNOWLEDGEMENTS

The following people from Hewlett-Packard contributed to the interface design and implementation of job control for HP-UX: Jim Barton, Dave Decot, Larry Dwyer, Jeff Glas-son, Rita Hanson, Stephen Hares, Steve Head, Bob Lenk, John Marvin, Dave Mears, Peter Notess, Arn Schaeffer, Eviatar Shafir.

Guy Harris from Sun Microsystems made many substantive comments and suggestions which contributed to the interface design and to this paper.

8. REFERENCES

- [ATT86] *System V Interface Definition*, Issue 2, AT&T, 1986.
- [Bach84] M. J. Bach and S. J. Buroff, "Multiprocessor UNIX Operating Systems", *AT&T Bell Lab. Tech. J.*, 63, No. 8 (October 1984), pp. 1733-1749.
- [Head85] Stephen Head and Donn Terry, "Reliable Signals Proposal", IEEE P1003 Proposal #P.042, Hewlett-Packard Co., September 11, 1985.
- [Joy80] William Joy, "An Introduction to the C Shell", Computer Science Division, University of California at Berkeley, November 1980.
- [Len86] David Lennert, Guy Harris, et. al., "System V Compatible BSD-style Job Control Facilities", IEEE P1003 Proposal #P.047, Hewlett-Packard Co. & Sun Microsystems, April 9, 1986.
- [Ritch79] Dennis M. Ritchie, "The UNIX I/O System", *UNIX Programmer's Manual*, Seventh Edition, Volume 2b, Bell Telephone Laboratories, Murray Hill, NJ, January 1979.

- [Roch85] Marc J. Rochkind, *Advanced UNIX Programming*, Englewood Cliffs, N.J.: Prentice-Hall, 1985.
- [Harris86] Guy Harris, "Notes on Signal, Terminal Interface, and User/Group ID Handling Proposals", IEEE P1003 Proposal #P.045, Sun Microsystems, January 11, 1986.
- [Thom78] K. Thompson, "UNIX Implementation", *Bell System Tech. J.*, 57, No. 6 (July - August 1978), pp. 1931-1946.
- [UCB83] *UNIX Programmer's Manual*, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, Computer Science Division, University of California at Berkeley, August 1983.

Unix as a Virtual Machine Environment

Robert E. (Rick) Genter
BBN Laboratories Inc.
10 Moulton St.
Cambridge, MA. 02238
rgenter@bbn.ARPA

Introduction

Over the past several years the Unix¹ operating system has gained wide acceptance in both academia and industry as the preferred environment for software development. Through the use of utilities such as the Portable C Compiler, *lint*, *make*, various debuggers (e.g. *adb*, *sdb*, and *dbx*) and source code control systems (SCCS and RCS) the software engineer may produce more, higher quality code in a shorter period of time than may be produced on other systems.

Unix is primarily designed for the development of *application* software, also referred to as tools. These tools are in turn designed for optimal application in a Unix or Unix-like environment. The purpose of this paper is to demonstrate that the standard Unix environment may also be used to develop *system* software; specifically, operating systems. No new tools are required for such development; rather, a set of methodologies is defined. A description of a virtual machine environment is presented, followed by a discussion of how Unix maps into this model. Two case histories are also discussed, as well as a hypothetical case involving running Unix on Unix.

What is a Virtual Machine?

A *virtual machine* is an environment which presents its applications with the illusion of executing directly upon a dedicated set of hardware. A true virtual machine environment can provide a complete set of virtual hardware. The virtual hardware provided falls into three distinct categories, the first of which is the virtual processor. The virtual processor can be either a subset of, a true copy of, or a superset of the underlying physical processor.

The nature of the virtual processor is dependent on the complexity of the virtual machine implementation. A VAX-11² virtual machine implementation, for example, may choose not to implement the virtual memory portion of the VAX architecture. This would be considered a subset implementation, as the full architecture of the underlying processor is not represented in the virtual machine.

Perhaps the most popular virtual machine implementation, IBM's VM/SP³, provides a true virtual machine implementation. Every feature of the underlying physical processor is available to the virtual machine, including full virtual memory and I/O capabilities.

An example of a superset implementation would be an Intel 8086 virtual machine implementation which included emulation of the Intel 8087 floating point coprocessor, whether the underlying physical hardware contained such a processor or not. In this case a portion of the hardware is being entirely simulated in software, providing a virtual machine of greater power than that provided by the physical hardware.

The virtual processor will naturally have memory. This memory is *not* virtual memory in the usual sense of the word, even though it is usually implemented through virtual memory machinery. The nature of the virtual processor's memory is dependent upon the virtual processor architecture. For example, the hypothetical 8086 implementation described above would not support virtual memory at the virtual machine level, due primarily to its lack of physical hardware support.

¹Unix is a trademark of AT & T Bell Laboratories

²DEC, VAX, and PDP are trademarks of Digital Equipment Corporation

³IBM, CMS and VM/SP are trademarks of International Business Machines Corporation

VM/SP, on the other hand, does provide virtual memory at the virtual processor level. An application running under VM/SP may initialize virtual page tables, enable address translation in the virtual program status word, and implement virtual memory on the virtual processor.

A true virtual machine implementation will also provide other virtual hardware along with the virtual processor. A virtual timer and a virtual time-of-day clock are usually provided. The virtual time-of-day clock is usually initialized to reflect the real time-of-day clock. This is not required, however. Thus a virtual machine operated by a user in one time zone may have its time-of-day clock set to a different time than one operated by a user in a different time zone. The virtual interval timer is also specific to the virtual machine; it can be used to measure real-time intervals, or virtual CPU time intervals.

A virtual machine environment may also provide a variety of virtual peripherals. Virtual peripherals exist in two varieties: mapped, and simulated. A mapped virtual peripheral simply maps to a corresponding physical device. A simulated virtual peripheral implements a peripheral which does not have a corresponding physical device.

VM/SP provides both types of peripherals. For example, a virtual tape device under VM/SP maps directly to a physical tape device. A virtual card reader, on the other hand, is usually implemented as a disk file with a spooling mechanism which "loads" the cards into the reader. A common means of communication between two virtual machines under VM/SP is for one machine to "punch" virtual cards to a spool file, which can then be read by the other machine's virtual card reader.

Mass storage devices under VM/SP can fall into both categories. Since VM/SP allows many virtual machines to be active simultaneously, it would be very inefficient to map a virtual disk drive directly to a physical disk drive. Thus VM/SP supports the concept of the *minidisk*. The minidisk is defined simply as a partition of a physical disk (not very different from Unix' file system partitions). Virtual machines may then mount individual minidisks for their use.

Under some circumstances, it is desirable to map a virtual disk to a physical disk, *e.g.*, when performing any sort of backup operation. VM/SP also supports the mapping of virtual disks to physical disk devices.

Naturally the user of a virtual machine needs some method of communicating with the virtual hardware. This is usually accomplished through a *control program*. The control program performs a number of purposes, including providing a "front panel" to the virtual machine. The virtual front panel is usually implemented as a command language interpreter running on the virtual machine's console (mapped to the user's terminal). The command language permits the user to specify the mapping of virtual devices to physical devices, the initial loading of application programs on the virtual processor, the examination of the state of the virtual processor, etc. The control program also provides the services necessary to implement the virtual processor.

A virtual processor is superficially implemented in the same manner as a process under 4.2 BSD Unix. A program executing on the virtual processor sees a contiguous block of memory starting at location 0. On a virtual machine, however, the program is started initially in supervisor mode. It is up to the program to perform any required initialization of the virtual processor, such as enabling interrupts, initializing page tables, probing for I/O devices, etc.

How can this work? What is to prevent a virtual machine from executing a HALT instruction, or performing some other similar disaster? The answer lies in the implementation of the control program. The application program, though executing in supervisor mode on the virtual processor, is executing in user mode on the physical processor. When a privileged instruction such as HALT is executed, the physical processor generates an exception or interrupt condition. The control program then handles the interrupt, performing any necessary decoding to determine the instruction which generated the interrupt and subsequent effect the execution of the instruction will have on the virtual processor, and modifies the state of the processor accordingly. In the case of a HALT instruction, the control program would most likely inform the user on the virtual console that the virtual processor has halted, then return to its command mode.

The control program is a part of the virtual machine environment and, as such, exists "outside" the address space of the virtual processor. An application executing on the virtual processor is not aware of the existence of the control program, nor can it normally communicate with it. Since there are cases where it is advantageous for an application to be able to communicate directly with the control program, a virtual machine implementation may implement new instructions, not normally present on the physical processor, which convey information directly to the control program. An example of this is the *cpdiag* instruction supported by VM/SP.

The advantages to operating in a virtual machine environment are obvious. In the event of a failure of the application, only the virtual hardware need be rebooted. Hardware not present in the physical system may be emulated in the virtual system. In general, the developer has greater control over the application than would be practical if it were executing directly on the underlying hardware.

There is one obvious disadvantage to operating in a virtual machine environment: performance. Each privileged instruction must be simulated by the control program, incurring a large overhead per instruction. Furthermore, all I/O instructions have an additional overhead associated with the mapping of the virtual device to the physical device, or with the simulation of the virtual device. For example, under VM/SP, if the instruction SIO X'191' is executed on the virtual processor (Start I/O on the device whose channel address is 0x191), the control program must first handle a privileged instruction exception (the SIO instruction is a privileged instruction on System/370 architectures), determine that the instruction is a SIO instruction, then look at the device on which I/O is attempting to be started, and then determine whether it is a mapped or simulated device. Finally, if the device is a simulated device (such as a minidisk or a virtual reader), the appropriate driver is invoked. If the device is a mapped device, the physical channel address for the device is determined, and a *real* SIO instruction is issued for the device⁴.

The underlying hardware architecture also provides constraints on the virtual machine implementation. Memory-mapped I/O is much more difficult to emulate than I/O performed through special processor instructions (*e.g.*, PDP-11 vs. System/370). On the latter, a privileged instruction exception is generated when an I/O is attempted. On the former, the memory management unit must be set such that an access to virtual I/O space will cause a trap, which can then be handled by the control program. If I/O space does not consist of a small number of small contiguous areas then it is probably impractical to implement any sort of I/O simulation for the virtual processor.

Processors without a separate "supervisor" mode are also difficult to impossible to emulate (*e.g.*, 8086 vs. VAX). Without a separate supervisor mode, it is impossible to prevent a virtual processor from executing an instruction which could corrupt the state of the physical processor or the control program.

Given the above characteristics of the virtual processor, it should be obvious that the typical application executed in a virtual machine environment is an operating system. Under VM/SP, IBM supplies several operating systems. One of its more popular interactive operating systems is CMS (Conversational Monitor System). CMS is relatively simple, supporting a single user executing a single process. Since VM/SP supports the coexistence of many virtual machines, CMS is not required to handle any of the tasks of a normal time-sharing, multitasking operating system.

Furthermore, different virtual machines may be executing *different operating systems*. For example, under VM/SP, each interactive virtual machine user may be executing a copy of CMS, whereas one virtual machine may be executing OS/MVS (a batch-oriented operating system). In fact, during the development of new versions of VM/SP, it is quite typical to execute VM/SP under VM/SP.⁵

⁴It is actually more complicated than this. Not only does the SIO instruction have to be scrutinized, but the channel program stored in the virtual processor's memory has to be duplicated in the memory of the physical processor and possibly interpreted.

⁵The performance of VM/SP under VM/SP is less than optimal.

The Unix Virtual Machine

Given the above description, how does Unix fit into the virtual machine model? First, let us describe the Unix virtual machine. Note that in the following discussion, features described are provided in terms of Version 7 Unix. Features specific to 4.2 BSD and System V may be used to enhance the virtual machine environment, however, dependence on such features may be avoided.

The Unix virtual processor supports the user-mode subset of the underlying physical processor. No provision is made for simulating privileged instructions; if an application executing on a Unix virtual processor executes a privileged or non-existent instruction, it receives the signal SIGILL.

The amount of memory available to a Unix virtual machine is variable. By using the *sbrk(2)* system call, the Unix virtual machine can extend (or shrink) the amount of memory available to it. Furthermore, Unix does not provide anything in the way of virtual memory support to its applications; *i.e.*, Unix does not provide a virtual memory management unit to the virtual processor.

The timer facilities available to the Unix virtual processor are dependent on the version of Unix. The basic support available on most implementations is provided by the *alarm(2)* system call. This gives an interval timer based on real time and accurate to 1-second resolution. Under 4.2 BSD there are three interval timers available: an interval timer based on elapsed "wall-clock" time (real-time), an interval timer based on CPU time (virtual-time), and an interval timer suitable for profiling program execution. In all cases, the expiration of the timer causes a signal to be generated to the application executing on the virtual processor (SIGALRM for real-time events, SIGVTALRM for virtual-time events, and SIGPROF for profile events).

The Unix virtual machine does not provide any special support for virtual I/O. Unix does partition its physical mass storage devices into logical partitions, however, these partitions are usurped by the Unix file system. Special permissions are required for an application program to directly access a disk partition. Most physical devices could be mapped to a corresponding virtual device one-to-one, but again, special permissions are usually required to directly access a physical device. 4.2 BSD Unix provides a simple form of virtual I/O through its pseudo-terminal device driver (*pty(4)*). Unfortunately this proves to be of little use to the class of operating systems most easily developed on top of Unix.

It seems that the Unix virtual machine is of little utility, riddled with limitations. What sort of operating system *can* be usefully executed on such a machine?

The answer is two-fold. First, the Unix virtual machine environment is best suited to developing *portable* operating systems. A portable operating system is one which is targeted to execute on a wide variety of processors or architectures rather than a single processor or architecture. By definition, portable operating systems tend to contain little hardware dependent code.

Second, the Unix virtual machine is well suited to developing *real-time* operating systems. At first this would seem to be a contradiction; Unix has a notorious reputation for poor performance, especially with respect to real-time computing. However, the Unix virtual machine is intended only for the development of the operating system, and is not its intended target environment. Real-time concerns are therefore not applicable.

The advantage of using Unix as a virtual machine environment for developing such a system is great. The presence of sophisticated software development tools greatly simplifies the coding and debugging phases of development. On VM/SP, for example, the development cycle consists of: boot the development OS (usually CMS), edit, compile, link, boot the test OS, debug. Debugging is limited to either the examination of a listing of a memory dump after the test OS fails, or the setting of machine language breakpoints and the examination of registers and memory during system operation.

On Unix there is no need to boot a development OS; Unix is that OS. "Booting" the test OS involves simply executing a Unix program. Furthermore, many operating systems are being developed

using high-level languages (C, Pascal, Ada⁶). Unix offers powerful source level debuggers in *sdb* and *dbx*, as well as offering *adb* for debugging at the machine language level.

Still, how can we overcome the limitations the Unix virtual machine presents to performing any serious operating system development? One answer involves a logical redefinition of the Unix virtual processor.

Just as the control program for VM/SP provides support for an artificial instruction (*cpdiag*), we can think of Unix system calls as a set of artificial instructions to be used to communicate with the “control program” on Unix, namely the Unix kernel. Using this redefinition, let's see how a portable real-time operating system might be implemented.

In general, an operating system implementation consists of a hardware dependent portion and a hardware independent portion. There are four areas which comprise the hardware dependent portion of the implementation. The first of these is system initialization.

Some of the specific tasks to be performed when a system initializes includes probing for available memory, probing for and initializing available peripheral devices, and finally setting up and initiating kernel processes. Under the Unix virtual machine, the first task, probing for available memory, is quite simple. On 4.2 BSD Unix, this involves two system calls: a call to *getrlimit*(2) to determine the amount of memory a process may allocate, and then a call to *sbrk*(2) to allocate all (or part) of it. On other versions of Unix, a binary search algorithm using successive *sbrk*(2) calls can be utilized to determine the amount of available memory.

The disadvantage to this technique is that it is very “hardware” dependent. Specifically, it depends on the *sbrk* and *getrlimit* “instructions” on the Unix virtual processor. Since it is unlikely that the target hardware supports such instructions, an alternate solution should be found.

Fortunately, through the use of function stubs, a mostly hardware independent solution can be implemented. Memory probing is usually performed by a loop which repeatedly accesses higher and higher memory locations until the access fails, thus determining the limit of available memory. Assuming that a call to *sbrk*(2) has already been performed to extend the size of the virtual processor's address space beyond that occupied by the operating system, the loop can be retained if structured as follows:

```
high_location = end_of_OS;
access_failed = FALSE;
catch_memory_fault (mark_failure);
while ( ! access_failed ) {
    high_location += increment;
    * (char *) high_location = '\125';
    if ( * (char *) high_location != '\125' )
        access_failed = TRUE;
```

The routine *mark_failure* simply sets the variable *access_failed* to TRUE and resets the memory fault condition. The only portion of code which becomes hardware dependent is the routine *catch_memory_fault*. On Unix, it consists of a simple call to *signal*(2) (*sigvec*(2) on 4.2 BSD). On the target system it will invariably become more complex, most likely setting up an interrupt handler which will save registers and in turn call *mark_failure*.

Probing for and initializing peripherals is less clear. Some devices may not be easily implementable on Unix. Some devices (such as a terminal device) might require no special initialization. The level of complexity is left to the developer. For example, a device such as an A/D (analog to digital)

⁶Ada is a trademark of the U.S. Department of Defense (AJPO)

⁷We use the value '\125' here because it is an alternating bit pattern (01010101) and not likely to be generated by device registers or other hardware that could otherwise masquerade as memory.

converter may be implemented as a stream of random numbers, or it may be implemented as a pregenerated Unix file. The initialization of this device might then be to seed the random number generator, or to open the Unix file.

Timer initialization is another situation which depends on the target hardware. If the target requires a simple interval timer with 1-second resolution, the *alarm(2)* system call can be used as the timer⁸. If more complicated timing is required, a set of simulation routines might need to be implemented. In the worst case, the full functionality of the target timer hardware may not be achievable on the Unix virtual machine.

The second area of hardware dependent code involves device drivers. Many of the more modern real-time systems support the concept of a device driver similar to that of Unix. As mentioned above, some devices such as a console terminal can be mapped directly to the corresponding Unix device; in this case the developer's terminal. Other devices which have no Unix analog, such as the aforementioned A/D converter, have to be completely simulated, but may be done so rather simply.

Some devices are not so simple, however. For example, some real-time systems support mass storage devices such as floppy or winchester disks to be used for high speed data collection. Some even implement a file system and use the disks for storing user data and program files. How can this be handled under Unix?

The solution is relatively simple. A Unix file can be preallocated and designated as a volume of the mass storage device. The driver for this device can still retain its structure. If the sections of the driver which talk directly to the hardware can be isolated into separate routines, then only these routines need be replaced when the operating system is ported from the Unix virtual machine to the target hardware.

The third area of hardware dependent code is that of context switching. When an operating system switches context between two processes, it needs to access the machine registers directly in order to save the registers of the old process and load the registers of the new process. The C library on Unix provides two routines which are almost adequate for this purpose: *setjmp(3)* and *longjmp(3)*.

The reason that they are "almost" adequate is that most implementations of *setjmp(3)* and *longjmp(3)* perform error checking which is not desirable when performing an operating system context switch. For example, under 4.2 BSD Unix, the stack pointer restored by *longjmp(3)* is checked to see that it is not below the current stack pointer. This usually indicates an attempt to transfer from a higher level routine to a lower level one, without setting up the intermediate stack frames. Normally this would be an error. In the case of an operating system, however, this is probably perfectly valid. All it means is that the stack area for one process has a higher memory address than the stack area for another process. Another problem with *setjmp(3)* and *longjmp(3)* is that, at least under 4.2 BSD Unix, they manipulate the signal mask. As we shall see below, this may not be a good thing to do.

The solution is to provide an alternate form of *setjmp(3)* and *longjmp(3)*. The alternate form of *setjmp(3)* should simply save the registers in the *jmp_buf* structure, while the alternate *longjmp(3)* simply restores them. The disadvantage to this approach is that machine language code needs to be written for the Unix virtual processor. If, however, the Unix machine uses the same underlying processor as the target environment (e.g., the Unix machine is a Sun-3 workstation while the target is another MC68020-based system), the resulting routines will also be usable in the target environment. In this case no effort or code is wasted.

The fourth area of hardware dependence in an operating system has to do with interrupt processing. On the target system there may be a number of interrupts which can be generated and which must be handled by the operating system. The handling of the interrupt may involve the execution of special processor instructions or the manipulation of "magic" memory addresses (interrupt vector addresses).

⁸Under 4.2 BSD Unix, the *alarm(2)* system call has been replaced with the *setitimer(2)* system call. *Setitimer(2)* allows finer resolution, down to 10 microseconds on a VAX.

On Unix, the virtual processor may be handed a signal, or a software interrupt. Handling the signal also involves the execution of a "special" hardware instruction; specifically, *signal(2)* (*sigvec(2)* on 4.2 BSD). If an extra layer of abstraction is used, this hardware dependency can be hidden from most of the operating system. Furthermore, the extra layer of abstraction need not incur any overhead, such as that of calling an extra function. For example, the routine *catch_memory_fault* mentioned in the memory probing routine above could be implemented as a simple preprocessor macro if the target is an MC68000:

```
#if UNIXVM
#define catch_memory_fault(rtn) \
{ \
    struct sigvec sv;\
\
    sv.sv_handler = rtn;\
    sv.sv_mask = 0;\
    sv.sv_onstack = 0;\
    (void) sigvec (SIGSEGV, & rtn, (struct sigvec *) NULL);\
}
#else /* ! UNIXVM */
#define catch_memory_fault(rtn) \
    * (unsigned long *) 0x8 = rtn;
#endif /* UNIXVM */
```

In the Unix virtual machine there is already one signal which must be handled by the operating system. This is SIGALRM, generated when the interval timer expires. As most operating systems manage some form of timer, the SIGALRM handler can be made to call the appropriate timer handler providing an adequate hardware emulation.

Can an adequate emulation be provided for other interrupts? Some Unix signals, such as SIGBUS (Bus Error) or SIGILL (Illegal Instruction) may be mapped directly to the corresponding interrupt handler in the operating system. For I/O devices, however, there is another problem.

Most versions of Unix do not provide for any form of asynchronous I/O. Thus it becomes impossible to emulate asynchronous I/O, alleviating the "need" for an I/O interrupt. 4.2 BSD Unix, however, does provide a limited form of asynchronous I/O support. Specifically, terminal devices may be placed in "non-blocking" mode through the *fcntl(2)* system call. Terminal devices can also be caused to send the signal SIGIO when they are available for I/O. SIGIO could then be mapped to the appropriate interrupt handler, probably through an intervening preprocessing function which would present the handler with a simulation of the appropriate system state. This method is still woefully inadequate, however. There is only one signal available for all non-blocking I/O, and only terminal devices (and IPC sockets) support asynchronous I/O.

Even though there are limitations, the Unix virtual machine environment *is* a viable entity. Let us examine two operating systems which actually execute under Unix.

Case history: CMOS

BBN has developed a portable operating system known as CMOS (*C Micro Operating System*). CMOS is a simple, fast, real-time system which is easily customized per application. On Unix CMOS can be represented as an archive; only the modules required by an application are linked into the final core image. CMOS also contains a module which performs all necessary initialization before initiating any user designated processes.

The hardware dependent portion of CMOS comprises only 5% of the code. By developing a Unix version of the hardware dependent code, the remaining 95% was debugged in the Unix environment. Furthermore, applications to be developed for CMOS were developed on the Unix version without knowing they were executing on the Unix machine (as opposed to the actual target machine).

CMOS provides a number of basic services, including memory allocation and deallocation, process initiation, termination and scheduling, interprocess communication through event queues, timer services, and I/O handling. CMOS does not contain a file system; most CMOS applications exist in ROM or are bootstrap loaded directly into memory. CMOS does not support swapping, paging, or any other form of disk-based memory.

Bringing CMOS up on Unix proved to be a simple task. The only portions of CMOS which required special handling on Unix were context switching, the implementation of a system clock, and system initialization. Context switching involved modified *setjmp(3)* and *longjmp(3)* routines as mentioned above.

The system clock was implemented using the elapsed CPU interval timer provided by 4.2 BSD Unix through *setitimer(2)* along with the SIGVTALRM signal. The elapsed CPU timer was used instead of the elapsed real-time interval timer so that behavior would be repeatable relative to the "system" clock; the same number of instructions always took the same amount of user CPU time.

System initialization used a memory probing method similar to that described above.

As CMOS is implemented as a library, using it under Unix is quite simple. CMOS applications are linked with the CMOS libraries, then executed. Applications specify a configuration file which is processed into tables of initial processes, devices, etc. CMOS contains the routine *main()*, thus it is allowed to perform its initialization before user processes are invoked.

CMOS applications can fail in two ways. First, like most operating systems, CMOS performs a number of internal consistency checks when a system service is requested. Should any of these checks fail a routine is called which displays a message on the console (the developer's terminal) explaining the failure, and then calls *abort(3)*. *Abort(3)* in turn generates a SIGIOT signal, causing a core dump to be produced. The user can then use a debugger to perform a post-mortem analysis and determine why the system failed.

The second way in which an application can fail is to directly cause a signal to be generated. Some operating systems may want to catch signals and try to handle them appropriately; CMOS lets a core dump be produced, again allowing for a post-mortem analysis.

Debugging a CMOS application turns out to be quite simple. Using the debugger⁹ the user can set a breakpoint anywhere within the application. Upon reaching the breakpoint, the user can request a stack trace. The stack trace will show a routine traceback *only for the currently active process*. This is because CMOS allocates a separate stack for each process. CMOS processes are created through a system service which executes in the system context, thus the stack shows the routines only for the current process plus some small portion of the system stack. This was an unexpected benefit of running CMOS under Unix.

Once the application appears to function correctly under CMOS on the Unix virtual machine, the application can be relinked with the target environment version of CMOS, transferred to that environment, and executed. The application may need to be recompiled if the target environment is a different processor than the Unix virtual machine.

Case history: AMBRE

Automatix Inc. has also developed a real-time operating system using the Unix virtual machine environment. AMBRE¹⁰ (Automatix Multi-Board Real-time Executive) is a general purpose real-time operating system. AMBRE supports many of the same concepts as Unix modified for the real-time environment. Indeed, AMBRE even supports a "Unix compatibility" library supporting emulations of most of the Version 7 system calls with some enhancements from 4.1 BSD and 4.2 BSD.

⁹The word debugger is used to refer to any of *adb*, *dbx*, or *sdb*.

¹⁰AMBRE is a trademark of Automatix Incorporated.

AMBRE provides a large number of facilities. It has facilities for memory allocation and deallocation, process control, a Unix-like signal mechanism, interprocess communications through both an event and a queue mechanism, I/O device access, and a hierarchical file system. AMBRE also supports systems with multiple processors, providing all of the above facilities to processes across all processors within the limitations of the hardware.¹¹

AMBRE is obviously more complex than CMOS. AMBRE more closely resembles Unix in that processes execute in user-mode, whereas the system kernel executes in supervisor mode. AMBRE is designed for the Motorola 68000 family of processors.

In CMOS system calls are implemented as library routines and the application executes in the same space as the operating system (supervisor space). In AMBRE, system calls are also implemented as library routines, but the library routines in turn invoke system traps to transfer into supervisor mode. Obviously, when executing on Unix, this can't occur. Thus the Unix version of AMBRE maintains a global variable which indicates whether the virtual machine is in "supervisor" mode or "user" mode. This variable thus functions as a virtual program status word, complementing the Unix virtual machine.

AMBRE, like CMOS and Unix, contains a set of device drivers and a standardized device driver interface. A good portion of each device driver is actually hardware independent, performing necessary bookkeeping functions. The hardware dependent portions were easily isolated, thus allowing the various devices to be emulated rather easily. Of greatest interest is the disk device driver.

AMBRE supports both floppy and winchester disks containing a hierarchical file system similar to that of Unix. On Unix, the disk device driver uses a predefined Unix file as its disk; if the file does not exist when AMBRE initializes, then the device is determined to be nonexistent at probe time and AMBRE claims to not have a disk. Otherwise AMBRE attempts to mount the disk as a filesystem disk (again, similar to Unix). To support running AMBRE on Unix, extra utilities were developed to generate and maintain preformatted file system "disks" in files¹².

The ability to use the AMBRE file system while running on Unix proved to be invaluable. It was quite simple to develop a series of file system exercise and debugging tools which could link with AMBRE on Unix and were used to debug the file system. By the time AMBRE was brought up on the target hardware, most of the system simply "worked the first time."

Naturally there were bugs uncovered when AMBRE was first ported to the target system. These bugs came from three different sources. First there were bugs in facilities that were not adequately tested while on Unix. Second were bugs in facilities that were not *testable* under Unix. Finally, bugs were found because the emulation performed on Unix did not reflect the actual behavior of the hardware on the target system. Even so, the number of bugs uncovered was small compared to what would be expected with an "untested" system. Again, the ability to use *dbx*(1) to debug the operating system was a big win.

A major portion of AMBRE which couldn't be tested in the Unix virtual machine environment was related to the multiprocessor aspects of AMBRE. Primarily this was because the development was performed under 4.2 BSD Unix. Had the development been performed under System V, the shared memory primitives supported by System V (*shmget*(2), *shmat*(2), *shmdt*(2), and *shmop*(2)) could have been used to implement the shared memory available to all processors, with one virtual machine/AMBRE kernel to a process.

¹¹ Some of the hardware configurations on which AMBRE runs do not have disks and therefore do not support file systems.

¹² In actuality, these utilities were written for AMBRE rather than Unix. Generating a disk, for example, simply involved accessing the disk as a raw device and writing the appropriate blocks necessary to create a blank file system. By writing the utilities for AMBRE, they were also accessible for the target hardware.

Hypothetical Case: Unix

So far we have examined two actual cases in which operating systems were developed and debugged in the Unix virtual machine environment. In both cases the only extra code which was required was a modified version of the *setjmp(3)* and *longjmp(3)* library routines, a slightly modified memory probe routine, and various hardware simulation routines. The amount of "throw-away" code generated represents a minimal cost to achieve such a dramatically improved development environment.

Suppose we wanted to implement a more complex system on the Unix virtual machine. Specifically, suppose we wanted to implement Unix on the Unix virtual machine. What extra work would be required?

The answer depends on the Unix to be developed and the target hardware. For example, to implement 4.2 BSD VAX Unix on a Unix virtual machine would require implementing a virtual disk device, virtual memory management hardware, a virtual network interface, and a complete virtual I/O subsystem. In addition, emulations for all privileged instructions would have to be provided. At first glance this would seem an overwhelming task and not possible without special kernel modifications. However, this is not the case.

The key is the use of Unix signals. An intelligent handler for the SIGILL instruction would be needed to decode the "illegal" instruction and determine what it really was, performing emulation or privileged instructions where necessary. An intelligent handler for the SIGSEGV signal would also be needed to detect attempts to access Unibus and MASSBUS I/O space, as well as normal invalid user accesses. However, none of the signal handlers would exist *in the process representing the virtual processor*. Instead, the virtual processor would be executed under the auspices of a parent "control program" through the *ptrace(2)* system call.

The inefficiency of such an implementation would severely limit its usefulness. In order to support an implementation of Unix on Unix, either a much simpler Unix would have to be implemented (*e.g.*, Version 7), or kernel modifications would have to be made to turn Unix into a true virtual machine operating system.

The kernel modifications required to make Unix a true virtual machine operating system would be in three areas. First, the virtual memory subsystem of 4.2 BSD (or System V Release 2) would have to be expanded to implement a virtual memory management unit per processor. Second, the I/O subsystem would have to be expanded to support device mapping and simulation. The *streams* concept from Eighth Edition Unix may be useful here. Finally, a privileged instruction emulation subsystem would have to be added.

On top of all this, most processes on Unix neither need nor want to run in a virtual processor environment. Thus the expanded facilities outlined above would have to be enabled on a per-process basis, with the default being disabled. This could be implemented as a loader option and a new "magic" number recognized by *exec(2)*.

Conclusion

By treating Unix as a virtual machine environment an entire class of operating systems may be developed without resorting to the often painful task of using assembly language monitors and hardware debugging tools for software debugging. Through proper structuring of the operating system, dependencies on the hardware (either the target or the Unix virtual machine) can be minimized thereby minimizing the job of porting from the Unix virtual machine to the target hardware. By making significant, yet manageable modifications to the kernel, Unix can graduate to a full virtual machine operating system, one capable of supporting running an instantiation of itself.

Acknowledgements

I would like to thank Debbie Deutsch and Muriel Prentice of BBN; Debbie for encouraging me to write this paper, and Muriel for doing 99.9% of the port of CMOS to Unix. Thanks also to Ofer Gneezy of Automatrix, who supported the development of AMBRE on Unix, and to Neil Webber of Automatrix, who performed a major portion of the design and development of AMBRE. Finally, thanks to Paul Morganthall of Information Resources Inc. for keeping me honest about VM/SP.

Some of the work described in this paper was performed for the Naval Electronics Systems Command under contract number N00039-85-C-0033.

References

Virtual Machine/System Product. Introduction, IBM Publication GC19-6200-3, Fourth Edition, 1984.

CMOS: The C Micro Operating System (Release 3), Ed Burke and Craig Partridge, BBN Communications Corporation, Tech Memo CC-0022, June 1984.

Unix User's Manual, 4.2 Berkeley Software Distribution, 1983.

VAX Architecture Handbook, Digital Equipment Corporation, 1981.

Porting UNIX[†] to a Network of Diskless Micros

- or -

UNIX on Tinfoil

W. Appelbe, D. Coleman, A. Fratkin, J. Hutchison, and W. J. Savitch

Electrical Engineering and Computer Sciences Department, C-014

&

Institute for Cognitive Science, C-015

University of California, San Diego

La Jolla, CA 92093

U.S.A.

ABSTRACT

UNIX is often criticized as unsuitable for inexpensive microcomputers because of its memory and disk resource requirements (UNIX 'thrives on big iron'). However, our experience has shown that it is feasible to implement a network of dozens of diskless microcomputers (IBM PCs) running UNIX with only a few file servers (IBM PC ATs) using a low-speed network. For typical loads, response time is comparable to that of standalone microcomputers with hard disks.

This article discusses how EMU (the Educational Microcomputer UNIX Network) was designed and developed, and its performance. In addition, we discuss our experiences building a high performance microcomputer UNIX network with limited hardware resources, and without UNIX kernel source.

1. INTRODUCTION

For the past four years a self-paced course in introductory Pascal programming at UCSD has been taught to hundreds of students each year, using a network of 60 IBM PCs, linked via a Corvus Omninet¹. During the past few years the host operating system, the UCSD p-System², has become less appropriate. The p-System is no longer used for upper-division Computer Science courses as it is insufficient for many applications, including compiler development, artificial intelligence, and graphics. UNIX has been adopted as the host operating system for almost all upper-division Computer Science courses because of its portability, widespread availability, and the broad range of software tools available. Students who took the introductory Pascal course had to learn another operating system, UNIX, afterwards.

Therefore, in June 1984 we undertook a project to migrate from the p-System to UNIX as a host operating system, for compatibility with other Computer Science courses and campus networks. With unlimited funds, we could have simply replaced the network of IBM PCs, but the

[†] UNIX is a trademark of Bell Laboratories.

¹ Omninet and Transporter are trademarks of Corvus Systems, Inc.

² UCSD Pascal and UCSD p-System are trademarks of the Regents of the University of California.

investment in hardware and self-paced software was too great. As a result, we decided to develop a UNIX network based largely on the existing hardware. The challenges presented by this project were formidable, in particular:

- Which IBM PC version of UNIX to choose to modify (PC/IX, Xenix, or Venix)?
- How to install the network drivers into UNIX without source for the operating system?
- How to obtain acceptable performance using only a few server machines with hard disk drives? (UNIX is notorious for its heavy use of disk I/O)
- What Pascal compiler to choose? (no version of PC UNIX came with a Pascal compiler, and teaching 'C' as a first language to non-Computer Science majors is sadistic)
- How to transport the existing software (written largely in UCSD Pascal) and used for self-paced testing and grading of students?

Our goals were:

- A UNIX kernel for each workstation
- A minimal set of UNIX utilities for interactive editing, compiling, and running small programs written in Pascal
- Good response time
- Low hardware and development costs

Fortunately we had a pool of experience with both UNIX and UCSD Pascal which enabled us to develop and test the software before the start of the fall term a year later. The network is fully operational, and is currently being documented and thoroughly tested under heavy loads. Software performance is good to excellent, largely due to the careful design and integration of the network software. The hardware costs were kept within design goals: four servers (IBM PC ATs) and 512k memory upgrades for each of the client PCs (to enable the editor, pascal compiler and interpreter, and kernel to be 'core-resident' in each client). A brief description of the Omninet hardware and the network's configuration can be found in Appendix A.

Overall, the project has been highly successful, and we hope to disseminate it as an example of a low-cost UNIX network for dedicated applications.

2. EMU's DESIGN

The overall software design was intended to consist of:

- A minimally modified UNIX kernel for clients, containing an Omninet driver so that most of the client's filesystem was mapped onto the server's hard disk
- A server that handled client requests
- An editor and Pascal compiler
- The automated quiz and bookkeeping systems
- Assorted utilities for remote machine and printer access

The software we had to start with was:

- Evaluation binary copies of PC/IX and Xenix for the PC XT
- Assembler source for an Omninet driver
- UCSD Pascal sources for some of the utilities we needed
- VAX C source for 4.2 BSD & System V

In an ideal world, the EMU Network would have been developed as a carefully integrated design, based on an initial analysis of possible options and interfaces. As with most projects that depend on interfacing unknown hardware and software, the development consisted of a sequence of design decisions made as we became aware of the interfacing problems. In general, our approach was to minimize the amount of new software developed. We hope others can learn as much from the evolution of the project as from the design of the final product.

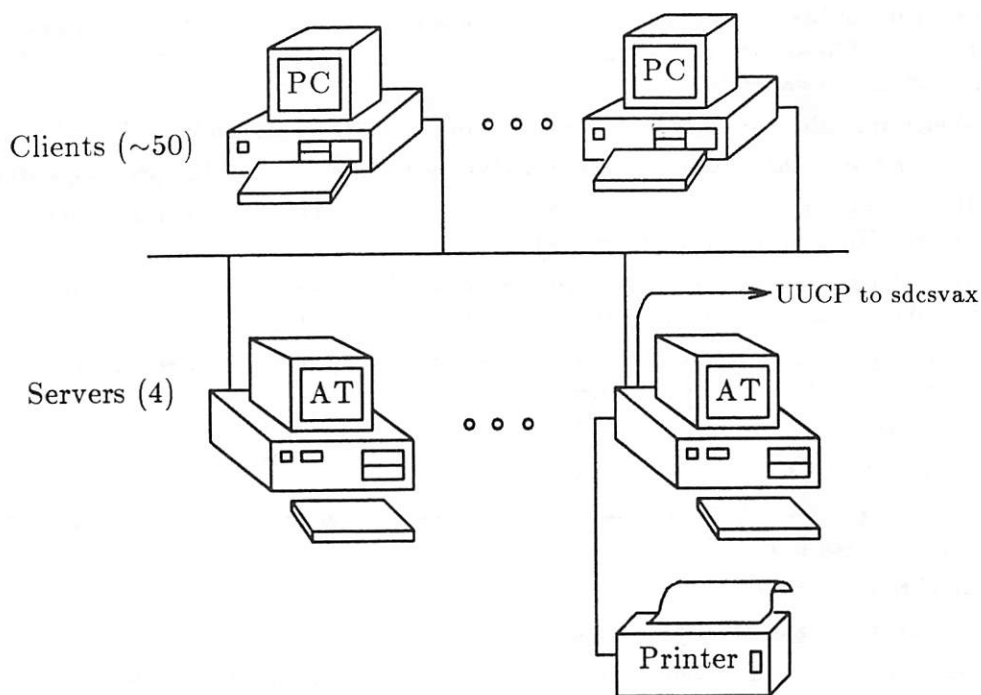


Diagram 1: The EMU Network Configuration

2.1. WHICH UNIX?

Several versions of UNIX were available for the PC. The ideal version for our project would be

- Adaptable — the kernel had to be modified for our network
- Efficient — good utilization of limited memory, disk, and CPU resources
- Complete — including tools and utilities for software development
- Inexpensive

At the time we undertook this project, there were three main implementations of UNIX for the IBM PC XT: Xenix³ marketed by The Santa Cruz Operation, PC/IX from IBM, and Venix/86⁴ from Venturecom. Xenix and PC/IX were the only candidates, as an evaluation copy of Venix/86 was not available.

The Santa Cruz Operation was helpful, and Xenix was a good system. It was Berkeley-ish, and came with many other utilities that we were used to having on 4.2 BSD systems. Xenix came with both **vi** and **csch**, and these were important to us. It did not come with kernel object files to link new kernels, although The Santa Cruz Operation offered to provide us with the necessary files.

PC/IX, on the other hand, seemed faster than Xenix. It started up programs faster, and compiled much faster. It came with kernel object files, and good documentation on writing device drivers and linking new kernels. But PC/IX is based on System III, so it did not have many of our favorite Berkeley utilities. This made it an unfriendly system. One advantage (we thought) was that it was supported and sanctioned by IBM. This was one of the main reasons we chose it over Xenix. We were eventually able to obtain ports of **vi** and **csch**. Other things we were accustomed to from 4.2 BSD, but were not provided with PC/IX (for example, **more**, **Mail**, **ls**, and **strings**) we ported ourselves.

³ Xenix is a trademark of Microsoft Corporation.

⁴ Venix/86 is a trademark of Venturecom.

PC/IX does have a couple of major advantages over any of the other PC UNIX systems we are aware of. First, it contains a kernel dump system, so a dump can be taken of a running, panicked, or damaged system. A **crash** utility uses the kernel dump information, and provides convenient access to all the system tables and stats. Second, PC/IX uses a contiguous buffer system. This is an attempt to make better use of the bandwidth of block devices. A field is added to the system buffer headers indicating how many contiguous buffers it represents. Contiguous buffers are buffers whose data portions are contiguous in memory, facilitating multi-block reads and writes. This reduces the number of read requests transmitted (most read requests are transmitted in units of 3 to 5 buffers), and this greatly improves the performance of our net disk system, since the server's overhead is reduced by a factor of 3 to 5 and the client only has to wait for 1 service delay, rather than 3 to 5 service delays.

In retrospect, it is hard to determine if we picked the right UNIX. We have had to deal with small kernels and limited available memory on the servers that would not have posed problems under Xenix. However, the increased disk throughput and better kernel debugging facilities of PC/IX are compensation. At the time of this decision, the PC AT had not yet been introduced. Had we known beforehand that it would run Xenix in protected mode and have a large memory capacity, we might have chosen Xenix. Now incompatible file systems would make it difficult for us run Xenix on the servers and PC/IX on the clients.

2.2. THE NETWORK

The structure of the network software on a server is illustrated below:

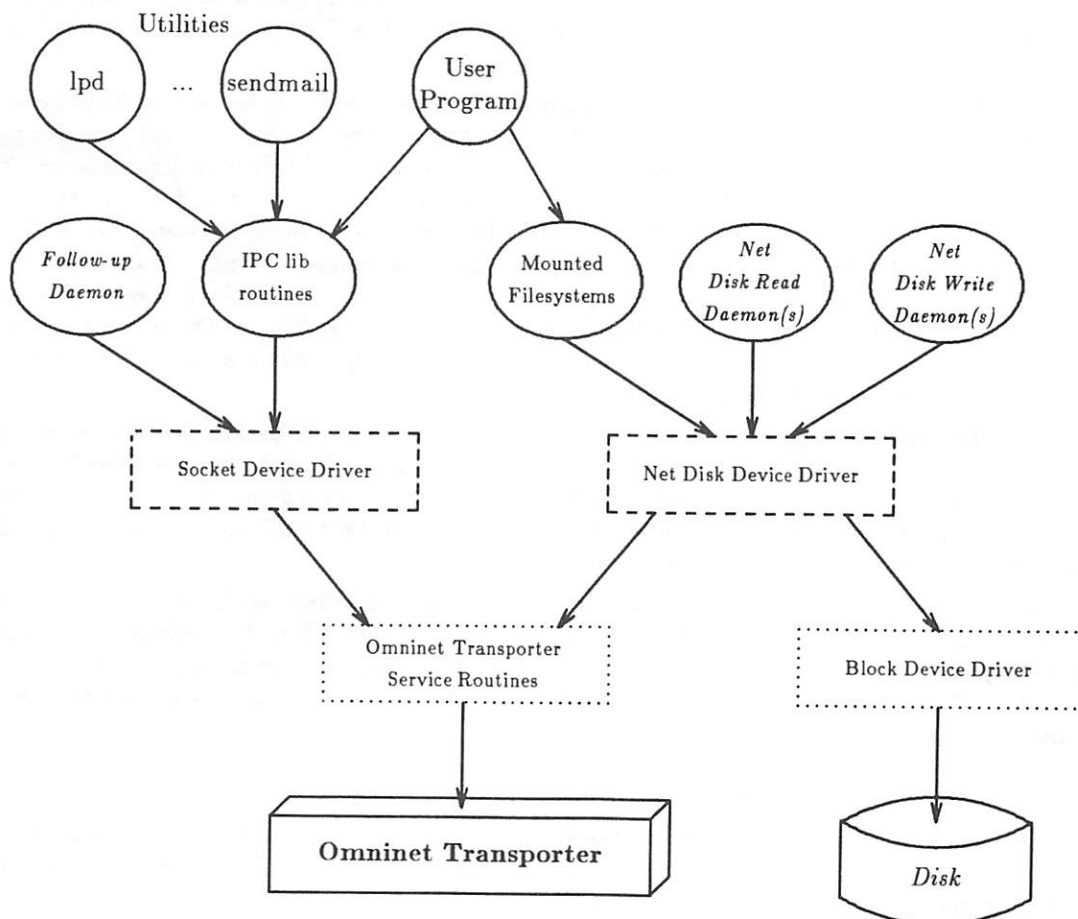


Diagram 2: EMU Network Client/Server Software Organization
(daemons and hard disk are not present on clients)

Because of financial constraints, the existing Omninet network hardware had to be retained. We had little experience with the Omninet Transporter, as those individuals who had previously developed the UCSD p-System's networking system had all graduated. We had the 8086 assembly language source for their design, but the p-System operating system is totally different from the UNIX kernel. So our first task was to attempt to get a feeling for how the Transporter worked. After looking over the assembly language sources for the UCSD p-System's networking system, it was decided to completely rewrite them, and to forge off with the Transporter Technical manual in hand.

2.2.1. THE CORVUS OMNINET TRANSPORTER

The Transporter does not fit cleanly into a multi-tasking kernel. The version we had does not provide any interrupts, so it has to be polled to see if anything has happened. We poll the Transporter once every 60th of a second from the timeout queue, and supplemented this by adding a few lines of assembly code to *idle()* so that the kernel continuously polls the Transporter when it is not busy doing anything else. During the start of a command to the Transporter, the address of the command must be 'handshaked' through a 1 byte hardware latch. Anytime during this synchronous handshake, the Transporter can have its attention drawn away (e.g., by an incoming packet), so it may not be ready for the next byte until quite a while later. A performance trade-off must be made, between waiting longer (in the hope that the Transporter will soon fetch the byte just stuffed), or letting other processes run for a while. Also there is no way of determining the current *state* of the Transporter, which makes debugging a driver difficult. The only way to tell that something has gone wrong is when the device stops responding (which is difficult to decide). Then the only possible solution is to cycle the power, forcing the Transporter into a known initial state.

The Omninet Transporter has one advantage over Ethernet⁵. It has a low level network layer protocol of its own, where it does its own retries, with exponential backoff, and provides positive acknowledgement that a packet was correctly received by the destination Transporter. The positive acknowledgement is useful, but the rest has not helped much. To understand the problem, it is necessary to have an idea of how the IBM PC Omninet Transporter works. First, it has four kbytes of on board memory. To perform a command, a command structure must be downloaded, including any data to send as a packet, one byte at a time through the port I/O system into this 4k, and then the three byte address of this command is handshaked through a single port latch. The Transporter then needs to be polled, by continuously reading a particular cell in the command structure just downloaded, for the result code to change.

The Transporter supports two basic operations: it can send a packet (to a host/mailbox address), and it can have up to four pending receives (one for each of the four *mailboxes*). Once the Transporter has received a packet for a mailbox (the received data going into the Transporter's local 4k), all packets for that mailbox are negatively acknowledged until the host tells the Transporter to again ready that mailbox for reception.

As long as a mailbox is setup to receive, everything works fine, as the Transporter's retry mechanism is designed to get through even a loaded network, but when the mailbox is not setup, the Transporter gets a negative acknowledgement, and sets an error code in its send command structure. The host then has to do its own retries, until the receiving host frees up the destination mailbox.

2.2.2. TRANSPORTER ENCAPSULATION

Our current Transporter interface presents a view of a somewhat different logical device for the higher protocols to deal with. There is a single *send()* routine, that simply sends a packet of a specified length⁶ to a specified host. *Send()* performs another level of retries, sufficient for

⁵ Ethernet is a trademark of Xerox Corporation.

⁶ The packet length can not exceed the receiver's mailbox length.

everything but the most loaded of receiving hosts. When it returns a failure, extreme conditions exist and only the higher level protocols know enough to do the 'right thing'. *Send()* can only be called by the top half of the kernel, as it will perform a *sleep()* when waiting for a packet transmission to complete. It would have been useful to implement *send()* as a simple queueing function so it could be called by the bottom half, but the quantity of the state it keeps around, the lack of good memory management facilities under PC/IX, and the limited need for such a system caused us to decide it wasn't worth the effort or the code space. There are multiple receive routines, and the received packet's type selects which *xxreceive()* should be called to process the packet.

2.2.3. INTERPROCESS COMMUNICATION

Our prototype Transporter interface was slightly different from that described above; it was a much closer model of many of the aspects of the Transporter. We were just interested in getting *something* up and running. The Transporter is able to support up to four 'channels' of information at one time (because it has four mailboxes — unique addresses for reception of packets), so our first design was based on the idea of a Transporter device, with four minor device numbers per host address, for a total of 64×4 , or 256 minor device numbers (which conveniently happens to fit in a minor device number)⁷. We did not plan on having that many special file entries in each client filesystem, as each client would only have to talk to a couple machines. This initial version was used to test our driver routines that manipulated the Transporter.

After a little use of our prototype network communication system, it became evident that we could do a lot better. The question was what protocols to adopt? TCP/IP seemed like an obvious candidate, as we had access to both the 4.2 BSD UNIX and the MS-DOS⁸ MIT/PC implementation of IP. This would have allowed us to do a lot of initial development under the much more powerful environment of a VAX running 4.2 BSD UNIX. But how in the (8086) world were we going to fit IP into a separate I/D space kernel, with only 12 kbytes available for additional code and no sources?

One solution would have been to implement IP in 8086 assembly language, using intersegment procedure calls, thus removing the code space limitation entirely. Another would be to have the kernel use a set of daemons to do all the work. Both of these were rejected after a little consideration. First, we had no wish to do a project of this size in assembly language, and we had no chance of completing the project on time doing so. Second, we had grave questions about how much of a performance penalty we would pay because of all the I/O and context switching resulting from doing all the work in a user daemon. We were going to depend on this system for all our IPC and most of our disk I/O, and we were already committed to using slow clients on a slow network, so we had to be extremely sensitive to any performance-limiting overhead we might be adding.

Our final decision was to implement two entirely new protocols. One was inspired by the 4.2 BSD socket system, and the other would resemble a smart disk controller's command language. This had two advantages. We could design them specifically for the Omninet Transporter, enabling us to squeeze every possible ounce of performance out of it. And since they did not have to be an inter-net networking system, we could leave a lot of IP's complexity out of our design, enabling us to fit it in the small amount of available code space. Our method was to use the minor device number of the socket device to refer to a *logical* connection, rather than a *physical* connection as in the previous design.

In 4.2 BSD UNIX, IPC routines were added as new kernel calls. We did not have any sources, so we created a large number of device *ioctl()* calls, each of which would perform the required operation, and then wrote a library that provides an interface similar to the one we are all used to on 4.2 BSD UNIX. We also moved some of the 'nice' but not absolutely required argument checking into these library routines. We have made constant tradeoffs on this project

⁷ Corvus's Omninet is limited to a maximum of 64 hosts.

⁸ MS-DOS is a trademark of Microsoft Corporation.

between features and code/data space. Since the standard kernel was already reasonably close to the 64 kbyte text ceiling, everything we added had to be carefully considered. Our solution was to move these functions out of the kernel.

Because of the nature of a UNIX kernel with limited memory, we decided we would not attempt to buffer a great amount of network data, so we made all our network IPC connections synchronous. If two processes are connected, and one attempts to send data to the other, it is blocked until the consumer empties the single block buffer allocated to it, and sends an explicit message to the producer process that it can now send up to one block of data. This did not allow us to provide a datagram service, but we do provide both a sequential-packet and a stream service. We even provide a two byte out-of-band message service. The only thing we were unable to provide was the *select()* operation. To provide this would have required us to disassemble vast amounts of PC/IX, and we've had enough of that.

We now have many network utilities that work over our network. Those that did not require *select()*, we ported from 4.2 BSD UNIX, the major examples being the line printer system (*lpr/lpd*) and *sendmail*. We wrote some almost from scratch, since they were heavily dependent on *select()*, including *rsh/rshd*. We also have a homegrown version of *rcp*. We have a version of *inetd*, but we have an *inetd* process for each daemon connection. We use *inetd* so we don't use up our limited swap space on clients with a bunch of rarely used, but reasonably large, daemons.

We can get a shell to run programs on remote machines with '*rsh remotemachine exec csh -i*', but this leaves a great deal to be desired. For example, since the remote end is not a tty, screen-oriented programs (e.g., *vi*), refuse to work. Pseudo ttys would be useful, but we don't have the space, nor do we have the source to implement streams (that might allow us to fit in a stream based pty-like system).

2.2.4. NET DISK SYSTEM

The net disk system allows a host to read/write another host's block devices, and can be used as if it were a real, local disk⁹. The *nd* system also has a partitioning system built in, so we can partition a single server partition into any number of client partitions. The client side of the system performs the partitioning, because the number of simultaneous partitions can be quite large¹⁰. This helps reduce the time spent by the server for servicing requests. The *nd* system has a local short circuit I/O system built in, so its partitioning features can be used during access to local devices.

On the server side of the net disk system, there are four types of daemons, read daemons, a write daemon, a big-write daemon, and a write-buffer daemon. There are no daemons on the client side. The read, write, and the big-write daemons each have a separate request queue, with requests being enqueued by the receive routine of the *nd* driver.

2.2.4.1. READ DAEMON

The read daemon (or daemons, since the number is configurable) simply takes a read-request from the read-request queue, performs a special *ioctl()* call to perform the actual sequence of *getblk()* calls or the single *xxstrategy()* call on the requested device, and then sends the resulting data to the requesting host. The read-request queue uses the same trick as the write-request queue, so that whenever the queue is full, the clients will retry until the read-request mailbox is again setup. The read requests are small and we have made the read queue large enough that it rarely fills. Read daemons are scheduled using a 'most-recently-used' strategy, to reduce the likelihood of daemons swapping.

⁹ When using a partition on a server's hard disk, any block device can be set up as an *nd* device. Setting up *nd* devices as the underlying physical device is possible, but hardly worthwhile.

¹⁰ Each client has a private partition for its root and swap areas, and some users have a private partition (mounted when the person logs in).

The clients timeout whenever the service of a read request has taken too long, and then retransmit the unfilled portion, so the clients are immune to their server crashing, at least as far as reads are concerned (~90% of our network activity).

2.2.4.2. WRITE DAEMON

The write daemon is small, as all it does is transform the queued request into a *bread()* or a *xxstrategy()* call, depending on whether the partition is shared or private. The write-request packets contain the data to be written (512 bytes).

2.2.4.3. BIG-WRITE DAEMON

The big-write daemon has a large data segment, as its purpose is to collect large write requests into a single local buffer, so a single disk write can be performed on it. When the client side of the nd driver notices that it is about to do a large write, for example, when swapping, it sends a big-write request to the destination host, and waits a little while for a response (containing a write request id). It then sends normal write-request packets (each with its 512 bytes of data) to the server. The big-write daemon can only service one request at a time, and it times-out quickly when waiting for the end of the data to be written to arrive. When a write request arrives with the current big-write request id, the nd *receive()* routine copies the data directly to the big-write daemon's write buffer. When the last data packet arrives, or when the time since the last big-write packet exceeds the allowed value, the big-write daemon performs the actual write. Any time either the client or server side of this transaction times out, the rest of this write request is handled just like a normal write request.

Observation of the EMU Network in operation motivated us to implement the big-write daemon. We knew writes were going to be expensive, but we did not feel this was critical, since the clients would all have local floppy disks for their temporary and user files. We knew the clients would swap, but we did not expect it to happen too often, so we did not worry about it originally. During the first months of student use we noticed extreme performance degradation whenever 2 or more machines started to swap. What was happening was that the head of the disk had to seek back and forth, alternating between the two swapping machine's swap spaces, performing a 512 byte write at each end of the bounce. When only one machine swaps, the head 'sits' on the same track, and the 512 byte writes involve no seek time.

2.2.4.4. WRITE-BUFFER DAEMON

This daemon attempts to keep a queue of a few empty buffers available for non big-write requests. The nd *receive()* routine just copies the non big-write request into one of the available buffers, and links the buffer into the write queue for the write daemon. The client *send()* routine knows about write-request packets, and will use a special Transporter mailbox that is only used for the reception of write-request packets. Anytime the system runs out of buffers to place the data into, a special flag is set so the low level Transporter interface will not setup the special mailbox. This will cause all transmissions to this mailbox to fail. Clients will resend their write requests until the server sets up that mailbox, which occurs as soon as a buffer is available. This process allows handshaking the successful completion of a write request to be avoided. It could cause problems if a server crashes while it is performing a write, but we consider the performance improvement worth the small risk. We have experimented with rebooting the servers at random times while they are serving clients, and the window of risk for major problems is so small that we do not encounter it during normal network operation. We have caused a set of clients to swap like mad and then reboot their server, but that is not the normal state of our network.

2.3. THE EMU UTILITIES

Students using our network are primarily editing, compiling, and running Pascal programs. Typically, they do nothing else. This created a need for a good editor and a fast Pascal compiler. PC/IX came complete with a menu-driven Rand-like editor called **INed**, but no Pascal compiler. We used a ported version of **vi** rather than the **INed** editor, since students would otherwise have

to learn **vi** for other Computer Science courses, and because we preferred **vi**.

The question of which Pascal compiler to use was a major stumbling block. We wanted a small, fast compiler with good error diagnostics, such as Borland's Turbo Pascal¹¹. Our requirements were unusual in that we wanted a compiler that would compile fast, but since the size of the student's programs would be so small, we did not really care how fast the compiled programs ran.

Although several implementations of Pascal existed for the MS-DOS operating system, none existed for PC/IX, and no company we could find had any interest in ever having one. However, we had the sources for the 4.2 BSD Berkeley Pascal system for the VAX (**pi** and **px**). **Pi** is the compiler and compiles into a pseudo code. **Px** then executes (interprets) the pseudo code. **Pi** is large, but compiles quickly. It has excellent error diagnostics and warnings. It is both a compiler and 'lint'er and has some error recovery (where the compiler tries to patch up the input and continue compiling). This makes it excellent for student use. **Px** is also large, and the execution is painfully slow for large programs, but for tiny programs it is entirely acceptable. Porting these became our only choice. Unfortunately, pointers and integers are 16 bits long on an IBM PC, and 32 bits on a VAX. The 4.2 BSD version of **pi** appeared to never have been **linted**, and thus had many machine dependencies. Attempting to convert it was tediously slow.

Fortunately, the 4.3 BSD version of **pi** had been re-written to pass **lint**. With minor changes, the new version compiled with few problems and has been mostly stable since. The moral of this story is clear: **never** try to port any software before it has cleared **lint** on the original host. Moreover, write portable software to begin with. Surprisingly, almost a complete implementation of **pi** fit on the PC; the only things missing are the new conformant array code and constant sets can not be constructed at compile-time. There was one major bug in **pi**. Even though most of the code had defines for either 16 bit or 32 bit addresses, array indexing always used 32 bit pointers. This required adding a new opcode to convert from the 32 bit pointer to a 16 bit value. **Px** has some machine language procedures, but once these were re-written for the 8086/8088 processor, **px** also ported without too much difficulty. A Pascal debugger (**pdx**) is available on the VAX and is almost completely ported to PC/IX.

The automated quiz software for self-paced learning is described in Appendix B.

3. EMU's PERFORMANCE

From the beginning it was clear that the Omninet could not support a 'traditional' UNIX load, with elite students at each PC forking off compiles while they edit or read netnews. Fortunately, the EMU Network is devoted to teaching introductory Pascal, so that students have no need for all that UNIX has to offer (and generally little desire to do more than enough work to get an 'A'). If the PCs have enough memory to ensure that the editor and Pascal compiler/interpreter can be core-resident, the performance of the network is less critical. We equipped every PC with as much memory as possible, raising them from 128k to their maximum of ~512k¹². The text segments of the editor (64k) and the compiler/interpreter (64k each) are core resident, and bumping their in-core reference count insures that they are not repeatedly swapped into the per-PC swap area on the server's disk.

3.1. BENCHMARKS

The performance of the system can best be gauged by comparing a PC using the net disk device to a PC XT with a local disk:

- **fsck** takes approximately 2.7 times as long on a client's remote server partition, as opposed to a local XT partition

¹¹ Turbo Pascal is a trademark of Borland.

¹² The cost of the memory upgrade, about \$120 per PC, is substantially less than a hard disk and controller per PC.

- **Mail** takes about 2-3 seconds to load and run on a PC XT, and 3 seconds on a client
- **vi** (version 2.13) takes 1 second to 'come-up' on a client, and 2-4 seconds on a PC XT (this is because **vi** is 'text-locked')

On our oscillation test (10 kbyte reads in a damped oscillation from one end of the partition to the other), the PC XT is approximately twice as fast as the client. On our simulated 'hacking' test (which runs **vi**, **ls**, **pi**, and **px** on a pascal test suite of ~100 files), the client and the PC XT were head-to-head (28.8 minutes for the client compared to 27.5 minutes for the PC XT). For our particular use pattern, with the proper executables text-locked (**vi**, **ls**, **pi**, and **px**) our clients are as fast as a PC XT with a local hard disk.

A PC AT sending net disk data to a client can load the network about 60% while it is actively sending (this comes out to 60 kbytes/sec.; since we often get two Transporter-level retries per packet sent, this really loads the network!). In practice, with ~50 clients active the network is not a major bottleneck, as network use is sparse. User files are on a local removable floppy, and frequently used text images are 'text-locked'.

As for the IPC system, two PC ATs just transferring data will load the network about 80% (that results in about 80 kbytes/sec.). A PC AT and a client just transferring data can load it about 50%; two clients about 25%.

3.2. RELIABILITY

Early versions of EMU suffered from various reliability problems, such as 'dropped' packets due to server, client, or hardware malfunction. One particularly obscure bug was the 'buffer cache consistency problem'. If a client reads a block shortly after it was written, the old version of the block may be obtained if the server had not yet written its buffer cache. Such potential problems are inherent in distributed UNIX systems.

The current version of EMU can be modestly described as 'robust', for example, death of a server during a transmission merely causes a client to loop waiting for the server's recovery. EMU's hardware is also fairly robust (the MTBF for clients is about two weeks). Network reconfiguration cannot be done 'on the fly', but is reasonably straightforward. If a server dies, all that needs to be done is to edit a single configuration file and run a program to generate the new per-client configuration files. The clients can then be rebooted using a new server. Multiple client configuration files were chosen for speed at boot time, the central file is needed to preserve unity and make (re-)configuration simpler.

4. CONCLUSION

In retrospect, most of our problems were caused by the complexity of the task of building network software and our lack of tools for testing and debugging (e.g., the lack of a PC/IX **syslog**, and the intransigence of Omninet). To a certain extent we have been plagued by occasional hardware reliability problems (PC components are not designed for 24hr/day student hackers). Given more funds, we could undoubtedly have built the system with less effort. However, the performance of the EMU Network justifies the effort. A system with comparable performance using 'off-the-shelf components' (e.g., PC XTs with PC Net) would be more than twice the cost of our configuration.

5. ACKNOWLEDGEMENTS

The Automated Quiz system was developed by Scott Cormode, after an initial design by Lawrence Hartsook. Walther De Petris developed the Programming Quiz system and the Tester. The Bookkeeping and Scheduling systems were written by Mark Lesperance. Tracy Higuchi and Roger Bly rebooted countless machines and kept the lab running and the students placated.

Special thanks to all.

6. REFERENCES

- [1] General Technical Information: Omninet Local Area Network Corvus Systems, Inc. San Jose 1984
- [2] PC/IX Users Manual, Version 1.1 IBM 1984
- [3] UCSD Pascal User's Manual Version IV.0 Softech Microsystems, Inc. 1981

Appendix A: Omninet and the EMU Network Configuration

Omninet is a shared-bus network using RS-422 twisted pair cables. The Omninet Transporter card provides the services of the bottom four layers of the ISO Seven Layer Network Model. The Transporter card performs up to ten retries automatically and has a parity bit to avoid duplicate messages. It uses Carrier-Sense Multiple Access (CSMA) protocol to arbitrate bus usage.

Each Transporter card is assigned one of 64 unique host addresses and, in addition, provides four distinct sub-addresses ('mailboxes') where messages may be addressed. Typically each mailbox is used for a different function. The instantaneous transmission speed of the Omninet is one Mbit per second.

We have 47 IBM PC clients distributed onto four PC AT servers. In addition, we have four PC XT machines used for development and one PC XT used to monitor the network. One of the PC XTs functions as our UUCP gateway to sdcsvax.

Of the 47 clients, 34 are available for general homework use, eight are devoted to automated and programming quizzes, two are reserved for proctor use, and one is reserved for floppy disk backups.

Each server has 40 Mb of hard disk storage. Twenty Mb is devoted to development on the server, and the other 20 Mb is used for clients. One Mb is given to each of 14 clients for their private root and swap partitions, with the remaining six Mb divided into three mountable file systems, /bin, /usr and /net. We place all of our network-related programs and files into /net rather than /etc.

Students are required to have a floppy disk to store their private files. The students' login shell is a special program that steps them through inserting their floppy into the disk drive, running `fsck`, and mounting their floppy disk on /tmp. By mounting their floppy on /tmp, all `vi` and `pi` temporary files stay on their own disk and help cut network load. The students' home directory is also on their floppy disk (/tmp/home). The login shell runs `csch` and unmounts the floppy after the student logs out.

The bootstrap (both a maintainer version that can boot into stand-alone mode, and the proctor version that can only boot into multi-user mode) resides on another floppy¹³. The /unix on the clients' disks only contains a name list, so students never have access to a /unix unless they steal a proctor/maintainer boot disk.

¹³ This practically eliminates the possibility of students obtaining a bootleg copy of any part of PC/IX.

Appendix B: Support Software for Self-Paced Instruction

Since the EMU Network's primary use is for a self-paced introductory Pascal course, several software packages have been designed and implemented to automate the course. These systems are an automated bookkeeper, a quiz administrator, a work scheduler and a trouble reporter.

The automated bookkeeping system is composed of two programs, **bookserv**, the database maintainer and record server, and **booker**, the client user interface. **Bookserv** keeps track of the students' progress, the proctors' hours worked and time sheets, and the administration and grading of quizzes. **Booker** is a menu-driven user interface enabling proctors to modify student records and clock in for their hours. This interface allows us to have password protected levels of access to the student's record.

The quiz administration system is principally divided into three programs: **tester**, **aquizzer**, and **pquizzer**. **Tester** logs the student in and communicates with the bookkeeping system to get information regarding the student (quiz being taken, unit number, class, etc.), then it executes either **aquizzer** or **pquizzer**, depending on the type of quiz being taken. Upon completion of the quiz, it returns the quiz results to **bookserv** to grade the quiz and update the student's record. **Aquizzer** is the automated quiz driver. Automated quizzes are a series of randomly-selected questions, some true/false and some requiring a short computation. There is a different set of questions for each chapter in the textbook. **Aquizzer** sets up the environment for the automated quiz and executes the separate object modules for each chapter. In a programming quiz, the student must write a simple routine given only the specifications. **Pquizzer** gives the student a controlled environment in which to write and debug his test routine. It begins by entering **vi** with a skeletal declaration of the routine, and the student must insert the solution. After exiting the editor, **pquizzer** invokes **pi** to interpret the routine. If compilation errors occur the student returns to **vi** to correct them. Otherwise, **pquizzer** runs the solution with the help of a small Pascal driver program. The Pascal driver makes several calls to the student's routine with randomly selected data, checking to be sure the solution is correct. If the solution is incorrect, the student is shown a description of the error and returns to **vi**. When the solution is finally correct, **pquizzer** informs **tester**. These three programs, **tester**, **aquizzer**, and **pquizzer** work with the bookkeeping system to administer quizzes in an efficient manner with little proctor supervision.

In order to control the working hours of our proctors, we have a work scheduling system that keeps track of the work schedule signups and communicates with the bookkeeping system to check the reported paid hours. This system is similar to the bookkeeping system in that there is a server **schedserv**, the database maintainer and multiple client server, and **scheduler**, the client user interface.

To complete our lab automation, we have an automated trouble report system known as **alog**. This menu-driven system enables the proctor to report any machine and/or system problem to the maintainer(s) of that subsystem. It simply creates a blank report that is tailored to the type of problem that is being reported and lets the user add the pertinent information. Then it mails the report to the responsible maintainer(s).

A State-wide UNIX¹ Distributed Computing System

Tom Truscott, Bob Warren, Kent Moat
Research Triangle Institute
P.O. Box 12194
Research Triangle Park, North Carolina 27709
(919) 541-6488
...!mcnc!rti-sel!{trt,rbw,kmoat}

1. Introduction

RTI is helping to establish a state-wide distributed computing system that connects seven universities and research institutions across the state of North Carolina. It is implemented using FREEDOMNET, a software system developed by RTI. The system provides users an extended UNIX environment with transparent access to remote files and devices, as well as the processing power of several super-minicomputers. The goals are to encourage cooperation and resource sharing, and to provide a handy test-bed for research into issues of distributed computing, particularly those involving heterogeneity and lack of central administrative control. This paper describes the underlying network, the user's view of the distributed system, some interesting problems that have been encountered, and some details of the implementation.

2. Network Background and Description

The Microelectronics Center of North Carolina (MCNC) was founded in 1981 to promote research and development into VLSI design, test, and fabrication. RTI is a participating institution (PI) of MCNC along with Duke University, North Carolina State University, the University of North Carolina at Chapel Hill and at Charlotte, and North Carolina A. and T. University. Figure 1 shows the distribution of PIs across the state of North Carolina. As part of its efforts to promote cooperation among researchers at these institutions, MCNC has built a state-wide microwave system for teleconferencing and teleclassing, as well as for high-speed data communications [MCNC].

The data communications system consists of 1.5M bps data links connecting a MCNC-supplied VAX 11/750 at each PI to VAX 8600s at MCNC using 56K bps synchronous interfaces. Each PI has its own local area network, usually an Ethernet, connecting a variety of different machines to the MCNC gateway VAX. There are currently more than 50 machines in the total network (including VAXes, Suns, a Gould PN9050, Convex C-1s, and 3B processors), running many variants of UNIX (including 4.2BSD, 4.3BSD, Ultrix 1.1, Ultrix 1.2, UTX-32 release 1.2, Sun UNIX release 2.0, Convex UNIX release 2.0, and System V.2).

¹ UNIX is a trademark of AT & T.

All machines use TCP/IP except for a (currently disjoint) subnet of 3B machines that use 3BNet. Figure 2 shows a representative portion of the RTI, MCNC, and Duke networks.

3. User's View of the Distributed System

The various systems are linked by FREEDOMNET, a software system developed at RTI, originally based on the Newcastle Connection [OVER]. FREEDOMNET makes it possible to treat each machine's file system as a sub-directory in a larger UNIX file system. FREEDOMNET extends the tree-structured directory mechanism of UNIX to connect the root directories of each computer logically into one large UNIX directory tree, a *super-tree*. The extended directory tree has a *super-root* under which the root directory of each interconnected computer is located. The system name of each computer is used to identify its root directory while the *super-root* remains nameless. A network of the RTI Gould PN9050 ('rtisel'), a MicroVAX II at Duke ('dukeuvax'), and a Convex C-1 at MCNC ('convex1') logically looks like Figure 3.

The concept of a *current working directory* in the UNIX user environment takes on a more explicit meaning under FREEDOMNET. The *current working directory* now refers to a directory on a specific computer. Before, it referred to a directory and implied there was only one computer it could be on. In Figure 3, the *current working directory* is on 'dukeuvax', and is indicated by the '.'.

Additionally, FREEDOMNET introduces the concept of a *current working root*. The *current working root* specifically states which computer's root is to be

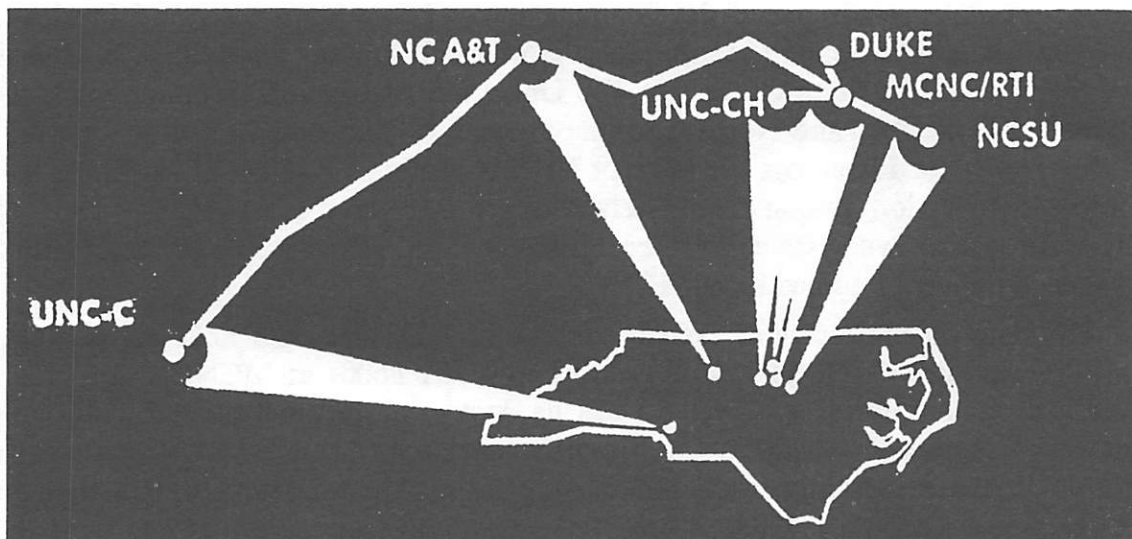


Figure 1. MCNC Microwave Network

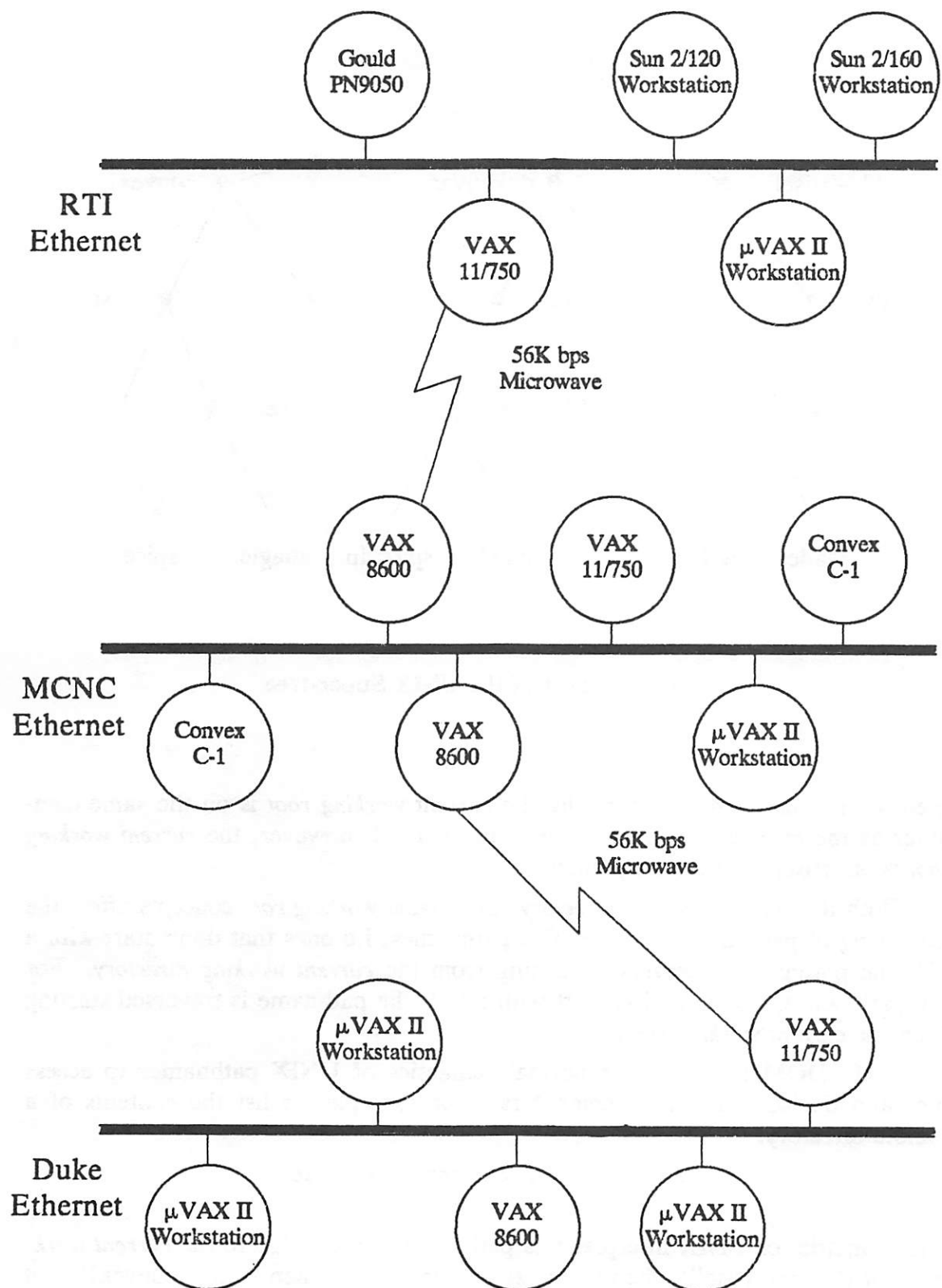


Figure 2. Part of RTI, MCNC, and Duke Networks

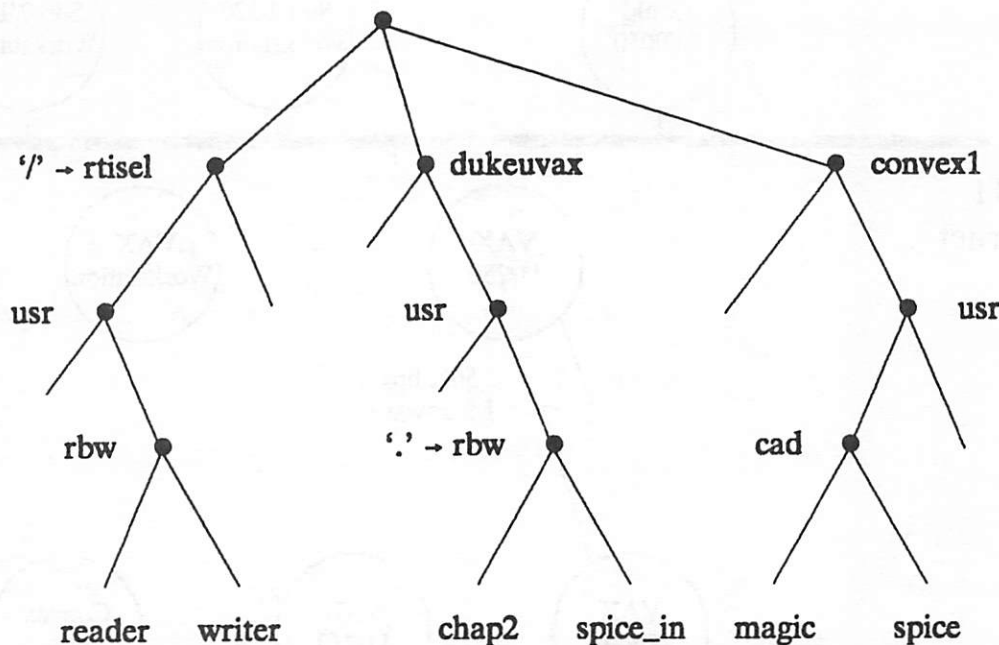


Figure 3. Part of the UNIX Super-tree

used for full pathnames. Normally the *current working root* is on the same computer as the *current working directory*. In Figure 3, however, the *current working root* is on 'rtisel', indicated by the '/'.

Both the *current working directory* and *current working root* concepts affect the traversing of pathnames. For relative pathnames, i.e. ones that don't start with a "/", the pathname is traversed starting from the *current working directory*. For full pathnames, i.e. ones that start with a "/", the pathname is traversed starting from the *current working root*.

FREEDOMNET uses the normal semantics of UNIX pathnames to access files and devices on remote computers. For example, to list the contents of a remote directory:

```
ls -l ../convex1/usr/cad
```

The semantics of UNIX interpret this pathname to mean "go to the *current working root* ("/" on 'rtisel'), then to the *super-root* (".."), then to the 'convex1' root ("convex1"), and finally to "usr/cad". Retaining UNIX semantics means that no new pathname symbols need to be memorized and understood. More

importantly, all user programs and UNIX commands remain portable across computers because the pathnames are semantically correct.

FREEDOMNET not only provides UNIX file operations over the network but all UNIX operations. This makes it possible to have more than just a distributed file system. Rather, FREEDOMNET implements a distributed *computing* system.

4. Benefits of Distributed Computing

The real benefit of a distributed computing system over a distributed file system is that one can not only share files, but all computing resources, including devices and CPUs. See also [USER].

4.1. Sharing Files

Users share files in cooperative projects, especially to facilitate writing papers. For example, a user located at RTI can edit a file in a directory on the Duke MicroVAX II using the commands

```
cd ../dukeuvax/usr/rbw
vi chap2
```

Note that the user changed his *current working directory* to the remote directory on 'dukeuvax', as in Figure 3. He could have also edited his remote file directly from 'rtisel' (where his *current working root* is located) by using the command

```
vi ../dukeuvax/usr/rbw/chap2
```

Of course, shell variables or symbolic links can be used to shorten path names as in

```
vi $dukerbw/chap2
```

where the shell variable "dukerbw" equals the string "../dukeuvax/usr/rbw". With appropriate permissions, a colleague at Duke can also work on the same chap2 file. FREEDOMNET provides access controls that work in conjunction with the standard UNIX file access permissions to make this type of file sharing easy.

4.2. Sharing Devices

Device sharing is equally transparent. For example, we have a Ramtek color graphics processor physically attached to an RTI VAX 11/750 known as 'rtivax'. We can run graphics software, such as the caesar VLSI layout program, on 'rtisel' by using symbolic links from the "/dev" directory on the Gould to the remote device entries:

```
ls -l /dev/ram*
lrwxrwxr-x 1 root ... /dev/ram -> ../../rtivax/dev/ram
lrwxrwxr-x 1 root ... /dev/ramtab -> ../../rtivax/dev/ttyha
```

The "/dev/ramtab" entry is for data table input and makes remote use of a serial line port on the VAX.

Similarly, only 'rtisel' at RTI has a 6250 bpi tape drive, but through the magic of distributed computing and symbolic links all machines have access to it. We feel the ability to share devices in a transparent manner should be an essential feature of any distributed computing system.

4.3. Sharing CPUs

Other researchers in the MCNC community need to execute large programs that require the vector processing capabilities of the MCNC Convex machines. FREEDOMNET allows a user to execute a program remotely on any interconnected computer. FREEDOMNET will usually execute a command or program on the computer where the command or program is found. For example, in Figure 3, the RTI user who has a SPICE file called `spice_in` in his *current working directory* on 'dukeuvax' can run the SPICE circuit analysis program on a Convex super-mini at MCNC using

```
../../convex1/usr/cad/spice < spice_in > /usr/rbw/spice_out
```

The `spice` program will execute on the 'convex1' system, take its standard input from `spice_in` in the *current working directory* on 'dukeuvax', and send its standard output to `spice_out` in "/usr/rbw" on 'rtisel' (the *current working root*), and its standard error to the user's terminal.

Commands executing remotely can be piped together just as if they were on the same computer. This has an advantage of spreading the processing load over different computers. For example, formatting a large document can be done with:

```
cd ../dukeuvax/usr/rbw
suntbl chap2 | vaxeqn | troff -me
```

where “suntbl” and “vaxeqn” have been set up using symbolic links with the commands

```
ln -s ../rtisun/usr/bin/tbl suntbl
ln -s ../rtivax/usr/bin/eqn vaxeqn
```

(‘rtisun’ is a Sun 2/160 at RTI). The three programs execute on the three different computers and pass data via pipes.

5. Administration Issues

Unfortunately, when crossing administrative boundaries the ability to “share and enjoy” conflicts with security considerations. The best we can do is provide controls that are as painless and minimal as possible [ADMN]. We use an access control scheme that optionally (but typically) uses the BSD-style `hosts.equiv/.rhosts` mechanism. It does not require identical password files among systems and can, if desired, allow for complete system equivalence (including `root`), or very restricted access. The local system administrator has complete control over which remote users are allowed access to his system. No central administration is required to add a new system or change individual user access privileges.

FREEDOMNET honors all access permissions of remote files by mapping a user’s local user ID/group IDs into appropriate remote user ID/group IDs while making remote accesses. If the remote system has no mapping for the user ID/group IDs, then it will deny all incoming accesses from that user. The definition of the mapping of IDs is kept and maintained by the system administrator of each remote system. Thus, each system administrator not only maintains control over all remote accesses to his system, but can also assign local user and group IDs independently of the other systems.

The most typical local mapping scheme is where the presence of a `hosts.equiv` file has the effect of mapping remote users and groups to local users and groups of the same name. This allows users with logins on administratively equivalent systems to transparently access all their files, both local and remote. In this case, `root` is usually mapped as well. System administrators can optionally permit anonymous accesses to their systems by allowing any otherwise unmapped users and groups to map into `FNanon`, a guest user ID.

FREEDOMNET will also inverse map user and group IDs. Inverse mapping provides a program with the local equivalent of a remote user or group ID. This allows programs to display the local symbolic user name or group name on listings

involving remote IDs. Inverse mapping is not used for permission checking or access control but is only for notational convenience. If there is no inverse map for a remote user or group ID, then the local `FNnotmap` ID is used. For example, a user lists a remote file belonging to `cad` of group `cadgrp` on the 'convex1' system:

```
cd ../convex1/usr/cad
ls -lg spice
-rwxr-xr-x 1 cad  FNnotmap 425984 Jan 24 9:34 spice
```

Although, the remote user ID, `cad`, is inverse mapped, the remote group ID, `cadgrp`, is not, so `FNnotmap` indicates an unknown inverse mapping.

An example of user/group ID mapping is shown in Figure 4. Here users `rbw`, `trt`, and `kmoat` are mapped between the Duke system and the RTI system. Note that their group IDs are different on the two systems. In fact, their user IDs are also different but their symbolic names as found in the local and remote password files are the same. There is no requirement that user/group IDs be the same on different systems, which is the normal case in administratively distinct systems. The RTI local system is set up so that all other incoming users are mapped to user/group ID `FNanon`, thus allowing anonymous access. From the Duke system, files on the RTI system owned by users other than `rbw`, `trt`, or `kmoat` will show up as owned by `FNnotmap`.

Some administrators have asked for more control over resource sharing than is normally provided by UNIX. For example, they would like to designate certain machines as "execute only" (such as a vector super-mini) and others as "file access only" (on cycle-starved machines). We are handling this with an extra `rwX` triplet provided for each remote user. Compute servers have a `--X` triplet for all users from remote institutions, while file servers have either `r--` or `rw-` set as desired.

../dukeuvax				local system			
<code>rbw</code> ,	102	<code><=></code>		<code>rbw</code> ,	127		
<code>trt</code> ,	103	<code><=></code>		<code>trt</code> ,	128		
<code>kmoat</code> ,	109	<code><=></code>		<code>kmoat</code> ,	138		
<code>*</code> ,	<code>*</code>	<code>-></code>		<code>FNanon</code> ,	<code>FNanon</code>		
<code>FNnotmap</code> ,	<code>FNnotmap</code>	<code><-</code>		<code>*</code> ,	<code>*</code>		

Figure 4. Example of user/group ID mapping

6. Heterogeneity Issues

The heterogeneous nature of our distributed system implies a number of obvious problems that must be hidden from the user. Among these are byte order and structure alignment differences. However, a harder problem is that of hiding operating system differences. In principal, the solution for UNIX is easy: emulate every foreign system call on the native system. In practice, it is a lot of work, especially to produce faithful and efficient emulations. FREEDOMNET has to deal with variant signal mechanisms, different sets of system calls, and provide correct remote directory access even in the presence of different hardware block sizes on the Gould and VAX. There are other problems such as mapping `ioctl` calls. These involve a fair amount of translation of the request numbers and information such as terminal speed.

The most interesting problems arise with binary programs and data. For example, there are a number of ways for a binary program for one machine type to end up on the file system of a different machine type. We extended `exec` to detect this and copy/execute the binary on a machine of the appropriate type.

Well-written UNIX software is "portable" in that it can be simply recompiled to run on different computer systems. Such software might not be "distributable" though, if it can not correctly read and write binary data from remote systems. This is more serious than it might appear; for example, running a "wrong" version of `newaliases` trashes the mail system since it operates on a hashed binary file! The fault is not so much with `newaliases`, but with UNIX itself for providing such inherently "undistributable" system calls as `read` and `write`. A distributable program will operate on binary data from many different computers, regardless of different byte orders, floating point formats, and structure alignments of the data.

The most common approach to making distributable software is to store all shared binary data under one external data representation. The shared binary data is converted from the external data representation to the native data representation of the accessing system by changing byte orders, converting floating point values, and realigning structures as needed. The shared binary data is converted every time the system accesses it, *even if the shared data is stored on the local system*. It is the excessive conversion overhead required for local processing that makes this approach unacceptable.

FREEDOMNET's approach to distributable software is to leave the data representation the same as the native representation of the system where the binary data is stored. Then a system can operate on local binary data without doing any data conversions, and only when binary data is transmitted to a remote system will it be converted to the remote system's native representation. Because no conversion is needed for local usage, local computing performance is undiminished. This approach even allows non-distributable programs to operate on local shared binary data without modification.

Since the differences in byte order, floating point formats, and structure alignments of remote binary data are handled during data transfers, new I/O system and library calls are required to indicate the necessary conversions. These new I/O calls are in addition to `read`, `fread`, `write`, etc., and are called `rti_read`, `rti_fread`, `rti_write`, etc. These new "Remote Transparent Interface" calls have an additional argument, called the *type-of* string, to specify the type of binary data being transferred (or seeked around) [PROG].

Using these new I/O calls, currently non-distributable programs can be made distributable. For example, Figure 5 gives a listing of two simple programs, a "writer" which writes a representative structure containing a float, 5 integers, and a character string to `stdout`, and a "reader" which reads the same structure from `stdin` and prints it out. If the programs are both run on the local system then a normal `fwrite` takes place. Otherwise the *type-of* string "{f5i8c}" is used to translate the data into the corresponding representation on the remote system. If these two programs are run locally, as in

```
cd /usr/rbw
writer | reader
```

then the result is, as expected,

```
2.71828 0 1 2 3 4 hello
```

If the writer is run on 'rtisel' and the reader run on 'rtivax' in an undistributable manner using `rsh`, as in

```
writer | rsh rtivax /usr/rbw/reader
```

the output is wrong since these two machines have different byte orders and different floating point formats:

```
1.71493e-13 0 16777216 33554432 50331648 67108864 hello
```

Only the integer zero and the character string are interpreted correctly. Using FREEDOMNET, the (more natural) command

```
writer | ../../rtivax/usr/rbw/reader
```

produces the expected results.

```

#include <stdio.h>
struct example {
    float    x;
    int      y[5];
    char     z[8];
} foo =
{
    2.7182818,
    0, 1, 2, 3, 4,
    "hello"
};

main()      /* writer.c */
{
    if (rti_fwrite((char *)&foo, sizeof(foo), 1,
        stdout, "{f5i8c}") != 1) {
        fprintf(stderr, "failed!\n");
        exit(1);
    }
}

#include <stdio.h>
struct example {
    float    x;
    int      y[5];
    char     z[8];
} foo;

main()      /* reader.c */
{
    int      i;
    if (rti_fread((char *)&foo, sizeof(foo), 1,
        stdin, "{f5i8c}") != 1) {
        fprintf(stderr, "failed!\n");
        exit(1);
    }
    printf("%g ", foo.x);
    for (i = 0; i < 5; i++)
        printf("%d ", foo.y[i]);
    printf("%s\n", foo.z);
}

```

Figure 5. Remote transparent interface example

The *type-of* string can support arbitrarily complex data structures (but not unions since they are rarely portable and are also non-distributable, or pointers since they have no meaning across machines). The rules for constructing the *type-of* string are straight-forward. However, we intend to enhance the C compiler to produce the appropriate string automatically or when an option is enabled. The compiler will thus be able to convert undistributable system calls to their distributable equivalents without explicit modifications to the program source. With this enhancement, for example, the kamikaze version of newaliases can be tamed merely by recompiling it (and libc.a, libdbm.a, ...).

7. Implementation

FREEDOMNET implements a distributed computing system at the system call level using a statefull server approach. There is a threefold advantage to implementation at the system call level: 1) the system call interface exists, is well documented, and is fairly portable; 2) realization at both the user and kernel levels is straight-forward; and 3) it provides a uniform interface for the system calls required for general remote execution, not just for those involving file access.

A statefull server avoids the non-UNIX file behavior inherent in stateless servers. In particular, a statefull server easily allows unlinking of open remote files, prevents the inadvertent unmounting of remote file systems containing open files, correctly performs guaranteed appends to remote files, correctly performs remote file locking for distributed processes, etc. Most importantly, a statefull system call server makes it possible to provide semantically correct behavior for all UNIX operations.

Figure 6 shows a schematic representation of where the FREEDOMNET layer resides in the distributed system. FREEDOMNET intercepts UNIX system calls within a local process, determines whether the file or device referenced is local or remote, and then either executes the system call locally, or executes it remotely via a remote server process. The remote server honors all remote access permissions by taking on remote user and group IDs that are appropriate for the local process.

7.1. Intercepting System Calls

FREEDOMNET intercepts system calls by substituting its own versions of these standard C library routines. Consequently, every program that uses FREEDOMNET must be relinked with the special FREEDOMNET version of the C library libc.a [INST]. Programs that are not linked with the FREEDOMNET C library continue to operate normally on their local system, and receive standard UNIX errors if they attempt to perform remote accesses. The (currently experimental) kernel implementation of FREEDOMNET intercepts all system calls inside the kernel, making relinking unnecessary. Versions of UNIX that

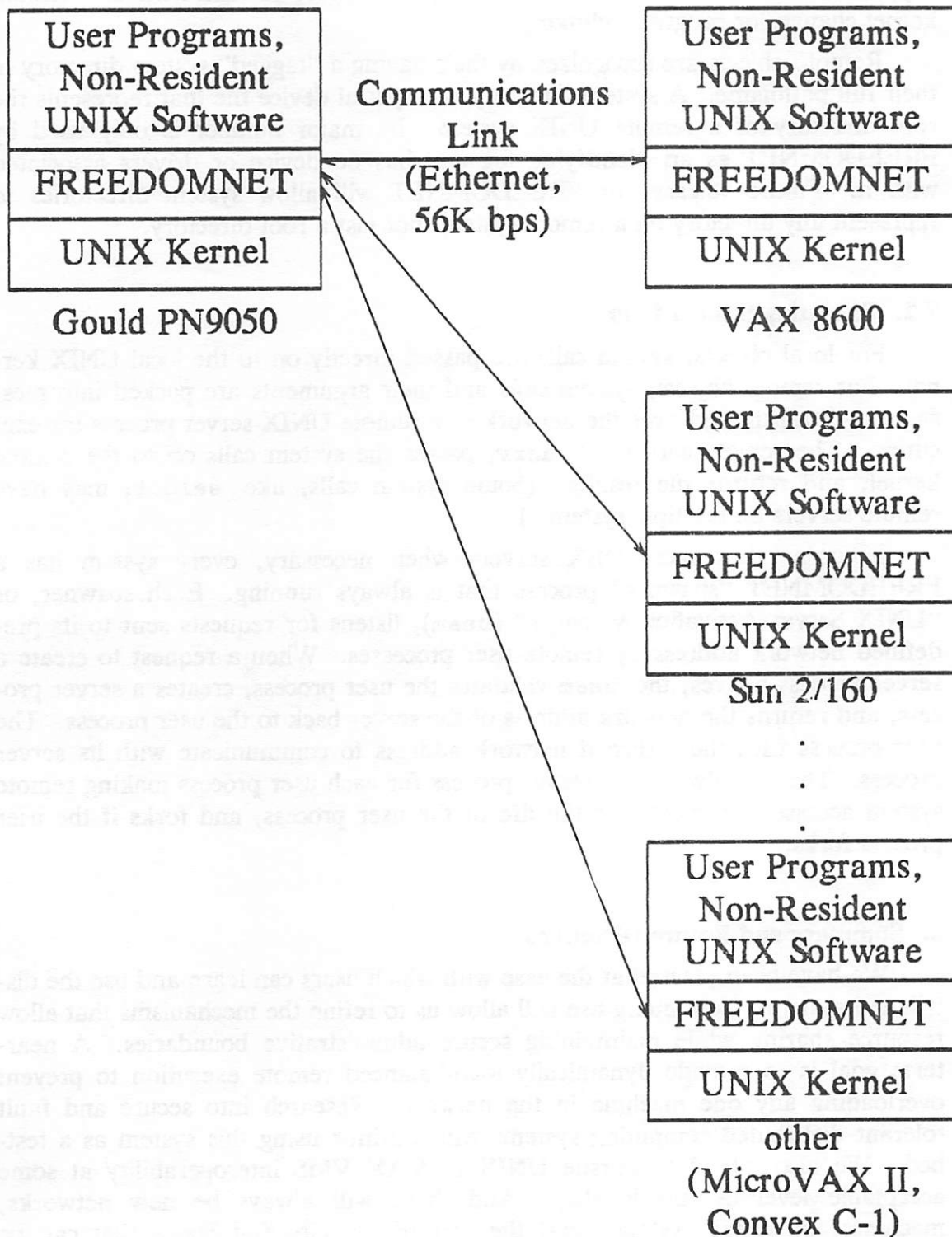


Figure 6. Logical View of the FREEDOMNET Layer

support shared libraries provide the ability to support FREEDOMNET without kernel changes or program relinking.

Remote objects are recognized by their having a "tagged" system directory in their full pathname. A system directory is a special device file that represents the root directory of a remote UNIX system. Its major number is only used by FREEDOMNET as an identifying tag and has no device or drivers associated with it. Future releases of FREEDOMNET will allow system directories to represent any directory on a remote system, not just a root directory.

7.2. Executing System Calls

For local objects, system calls are passed directly on to the local UNIX kernel. For remote objects, system calls and their arguments are packed into messages and transmitted over the network to a remote UNIX server process for execution. The remote server, or `usrv`, passes the system calls on to the remote kernel, and returns the results. (Some system calls, like `select`, may have remote servers on multiple systems.)

In order to create UNIX servers when necessary, every system has a FREEDOMNET "spawner" process that is always running. Each spawner, or "UNIX Server Activation Manager" (`usam`), listens for requests sent to its pre-defined network address by remote user processes. When a request to create a server process arrives, the `usam` validates the user process, creates a server process, and returns the network address of the server back to the user process. The user process uses the returned network address to communicate with its server process. There is always one server process for each user process making remote system accesses. It exists for the life of the user process, and forks if the user process forks.

8. Summary and Future Directions

We have been pleased at the ease with which users can learn and use the distributed system. Continuing use will allow us to refine the mechanisms that allow resource sharing while maintaining secure administrative boundaries. A near-term goal is to provide dynamically load-balanced remote execution to prevent overloading any one machine in the network. Research into secure and fault tolerant distributed computing systems will continue using this system as a test-bed. We also intend to pursue UNIX to VAX VMS interoperability at some acceptable level of functionality. And there will always be new networks, machines, operating systems, and the rest of the UNIXed States that can be incorporated into the distributed system!

APPENDIX 1 - References

- [MCNC]** Communications News, April, 1986.
- [OVER]** FREEDOMNET Overview: Distributed Computing in a Heterogeneous UNIX Environment, February, 1986, Center for Digital Systems Research, Research Triangle Institute, Research Triangle Park, NC.
- [USER]** FREEDOMNET User Guide, April, 1986, Center for Digital Systems Research, Research Triangle Institute, Research Triangle Park, NC.
- [ADMN]** FREEDOMNET Administrator Guide, April, 1986, Center for Digital Systems Research, Research Triangle Institute, Research Triangle Park, NC.
- [PROG]** FREEDOMNET Programmer Guide, April, 1986, Center for Digital Systems Research, Research Triangle Institute, Research Triangle Park, NC.
- [INST]** FREEDOMNET Installation Guide, April, 1986, Center for Digital Systems Research, Research Triangle Institute, Research Triangle Park, NC.

UNIX Based Distributed Printing in a Diverse Environment

William E. Johnston

Dennis E. Hall

Advanced Development Projects

Lawrence Berkeley Laboratory

University of California

Berkeley, California, 94720

wejohnston@lbl.arpa, ...ucbvax!lbl-csam!johnston

dehall@lbl.arpa, ...ucbvax!lbl-csam!hall

ABSTRACT

This paper presents our experiences using the Berkeley UNIX* line printer spooler mechanism (*lpd*, et al) to provide distributed, mostly laser printer based, typesetting and graphics output to a geographically dispersed, heterogeneous, set of host computers and users. The user interface is the usual set of UNIX commands, though the methodology employed is somewhat different from the usual. The user's environment is resolved on the local machine and the tasks of document formatting and device driving are relegated to dedicated server systems in order to remove these compute intensive tasks from the timesharing client, or user, systems. The details of the system are described, together with an analysis of performance issues, the suitability of *lpd*, and operational aspects.

1. Introduction

The Advanced Development Projects group evaluates, tests, and installs new computing technologies for Lawrence Berkeley Laboratory's Computing Division. Our primary function is to bring promising new computing technology into the scientific computing environment. This article describes one such project, a distributed printing facility.

Lawrence Berkeley Laboratory (LBL) is a multi-purpose research facility with programs in physics, astrophysics, nuclear chemistry, materials science, biophysics, research medicine, electron microscopy, mathematics, computer science, earth sciences, and renewable resources. These programs are supported by computing facilities consisting of a central VMS cluster of five VAX-8600's and several 780's, four UNIX systems, and a significant part of a remote Cray XMP. In addition to these "central" facilities, various other departments at LBL operate another twenty to thirty VAX's, an ELXSI, at least one Pyramid, and an indeterminate number of workstations. Most of these systems (except the Cray, which is a remote facility) are directly connected to a site wide

* "UNIX" refers to 4.2bsd UNIX, unless otherwise noted.

The work presented in this paper is supported by the U.S. Department of Energy under contract DE-AC03-76SF00098. Any conclusions or opinions, or implied approval or disapproval of a company or product name are solely those of the authors and not necessarily those of The Regents of the University of California, the Lawrence Berkeley Laboratory, or the U.S. Department of Energy.

Ethernet, or have some other direct access to the LBL internet.

One thing that most LBL computer users need is document formatting and hard-copy graphics output. These needs have been met, in part, by establishing a distributed printing system that handles, in various degrees, ASCII, \TeX , *troff*, Tektronix and UNIX *plot* format files. These various formats enjoy differing degrees of sophistication in their handling by the distributed printing system, and in the available output devices.

By way of review, recall that the batch mode text formatting typical of *troff* and \TeX may be represented logically as:

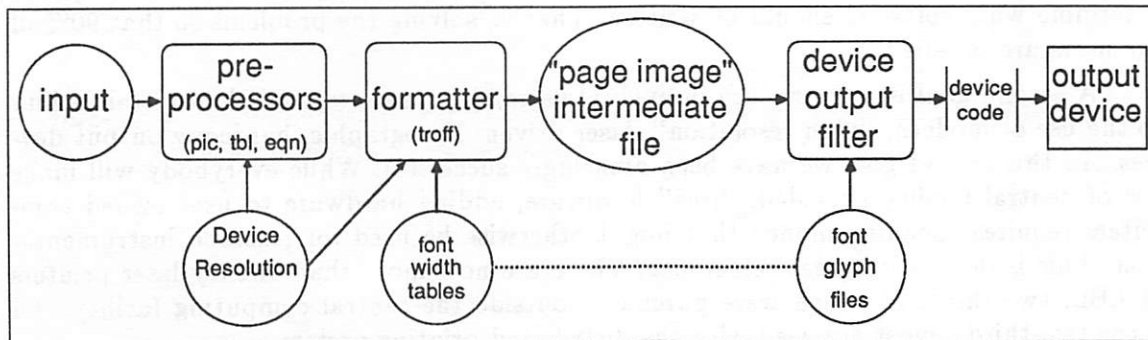


Figure 1

Most of the software development effort has gone into the remote *troff* service, whereby all user systems can spool *troff* input files to the back end servers. The *troff* input files have local file environment dependencies removed on the user system. The server systems are (typically) Integrated Solutions, 68020 based, UNIX systems dedicated to printing. The server systems execute *troff*, and its preprocessors (*eqn*, *tbl*, etc.), and any filters necessary to generate output device specific code. The server sends that code to the requested device and records accounting information. By far the most commonly used output devices are Imagen laser printers.

The universal availability of the distributed printing system within LBL depends on each user system being able to send input files to a server system, where it is received by *lpd*. In general this is done with an implementation of *lpr/lpd* on the user system, though variations have been tried. Access to output devices, which are located throughout LBL, is via Ethernet, long haul serial line, or some combination of these together with an AppleTalk network. (The site is a little more than one square mile of rough hillside which inhibits the use of local microware systems.)

The system is successful, and has been in operation for about 18 months. There are six public Imagen 8/300 printers; three of these consistently output 30,000 pages per month each, two output 20,000 pages per month each, and one outputs 10,000-15,000 pages per month. Three more public printers have been recently installed. Additionally, there are about ten private printers that output approximately 25% the amount of the

The following trademarks are acknowledged (apologies to anyone I missed): AT&T, Bell Laboratories: UNIX; Adobe Systems: *Transcript*, *Postscript*; American Mathematical Society: \TeX ; Apple Computer: *AppleTalk*, *LaserWriter*, *Macintosh*; Cray Research: *Cray XMP*; Digital Equipment Corporation: *DEC*, *VAX*, *VMS*, *DECNET*, *Q-bus*; ELXSI; ISSCO: *TELL-A-GRAF*, *DISSPLA*; Imagen; *8/300*, *Inovator*, *imPRESS*; Integrated Solutions, Inc.; Motorola: *68010*, *68020*; Mergenthaler-Linotype: *Linotronic-101*; Pyramid; Quality Micro Systems: *QMS*, *QUIC*; Sequent Computer Systems: *Balance 8000*; SUN Microsystems: *SUN*; Talaris Systems; Tektronix: *PLOT-10*, *TCS*; Versatec: *V-80*; Wollongong: *Eunice*; Xerox: *Ethernet*, *XNS*.

public printers.

2. The Printing System

2.1. Goals

The primary goal of the distributed printing system is to accept all common types of input files, and to print these files on any of the available output devices. It is recognized that a full connection of $N_{\text{formats}} \times M_{\text{output_devices}}$ is a goal that has to be tempered by the reality of available input and output filters, and the available software effort to generate new ones. We have been successful by applying the "90%" principle to determine what software should be written. That is, solving the problems so that 90% of the users are satisfied.

A second goal is to introduce individual users, departments, and the central facility to the use of modern, "high resolution", laser driven, Xerographic, hardcopy output devices. In this second goal we have been amazingly successful. While everybody will make use of central facility provided, "free" hardware, adding hardware to user owned computers requires spending money that might otherwise be used for research instrumentation. This is done with great reluctance. There are now more than twenty laser printers at LBL, two-thirds of which were purchased outside the central computing facility. Of those two-thirds, most are used with the distributed printing system.

A third goal is to make the system as operator-less as possible, a point to be discussed later.

2.2. The Mechanism for Remote *troff*

Of the several models of processing used by the various types of input files, the off loading of *troff* input to a back end server system, and output to Imagen printers, constitutes the major use of the distributed printing system. The mechanism for *troff* is described here. The mechanism for other input file types is described briefly in the section below on input types. In what follows, program and file names are typeset in *italic* in the text, and/or bold in the standouts. More detailed information may be found in reference [1].

The essential idea of the distributed printing system is to use *lpd*'s ability to route files to remote systems, and to use its flexible notion about output filters and the */etc/printcap* configuration mechanism, to send *troff* input files to a server system for processing and then to an output device.

The "user" or "client" system is the computer initiating a printing request. The "server" system is the computer that executes the programs needed to produce output. The user system may also be the server for a particular device, though usually not. There are normally many user system computers, and one server system for a given output device.

The distributed printing mechanism is just the sequence of processes through which data pass on their way from user systems to server systems. For remote *troff* targeted for an Imagen printer (*itroff*), the sequence is:

```
itroff → lpr → (user spool directory) → lpd →  
→ (network communication) → lpd →  
→ (server machine spool directory) → lpd → itroffd →  
→ icat.e → tbl | eqn | troff | catimp | ies
```

A brief description of this sequence follows.

2.2.1. Itroff

Itroff is the user interface for *troff* typesetting on the Imagen printers. For example, invoking *itroff -Pip1 troff.file* causes *itroff* to accept the user (*troff*) input file, process it through *soelim* to eliminate environmental dependencies (e.g., user include files), and then pass the file to *lpr* with flags giving the requested output device (*ip1*), and input file type (*troff*).

2.2.2. Lpr and the Spool Directory

Lpr takes a file and device specification, sends the input data file (and an associated control file which *lpd* creates) to a spool directory. The control file contains a variety of information: what output filter should be used to process the file; the user name; the user computer system name; etc. *Lpr* uses the file */etc/printcap* to determine where the spool directory is located, which system is the server for the requested printer, and how the user file should be processed. For example

```
lpr -t -Pip1 file.troff
```

selects the following sets of lines from */etc/printcap* (abbreviated in this example):

On a server system:

```
ip1:
    :sd=/usr/spool/imagen/ip1:
    :tf=/usr/local/imagen/bin/itroffd.ip1:
```

Or, on a user system:

```
ip1:
    :rm=host2:
    :rp=ip1:
    :sd=/usr/spool/imagen/ip1:
    :mx#1000:
```

In the first case (on the server system) the spool (or queuing) directory is specified (*:sd*), together with the filter to be used to process the file (*:tf*, in the case of a *troff* file).

In the second case (on the user system) the specification indicates what remote system to send the file to (*:rm*), what device to use for output (*:rp*, which may, incidentally, be different from the user requested device), where to spool (enqueue) the file (*:sd*), and a limitation on the size of files so processed (*:mx*).

In both cases *lpr* builds the control file (with a name of the form *cf** in the spool directory) which it sends, along with the input file (now with a name of the form *df**), to the spool directory (queue) where *lpd* takes over.

2.2.3. Lpd

Lpd is the worker back end for *lpr*. *Lpd* is a "daemon" process that watches the queues (as defined by *printcap*) for files needing to be processed. When a job shows up (a *cf** and *df** file combination) *lpd* looks at the *cf** file, which contains a pointer to an */etc/printcap* entry. The *printcap* entry specifies an output filter to be invoked on the data (*df**) file. For the Imagens, the output filter is a shell script called *itroffd.ipX* ("X" indicating the specific Imagen).

For a user system, *lpr* places the *cf** and *df** files in the local spool directory. The files stay in the local spool directory only long enough for *lpd* to forward them to the server system. Unless queuing is disabled for the requested device on the server, the files are normally forwarded immediately. For a server system, the files remain in the spool

directory until output processing is complete.

2.2.4. Itroffd.ipX

Itroffd.ipX is the *lpd* output filter for the Imagens. *Itroffd* sets up the sequence where the work is done :

tbl | eqn | troff | catimp | ies

The first three are familiar. *Tbl* and *eqn* were included in the output processing mostly to minimize the support needed on the user systems. Both of these preprocessors need to know various things about the output device.

Catimp converts the *troff* output to imPRESS (the language of the Imagen), and manages the loading of the character bit maps into the Imagen. *Ies* manages the network interface, transmitting data (imPRESS instructions) to the Imagen over a TCP circuit, and receiving status information back via UDP packets. For a serial line connected printer, *ies* is replaced by *ips*, which implements a sequenced packet protocol over a serial line.

2.3. Limitations of this Mechanism

This whole mechanism is to a certain extent a "90%" solution. Users who make use of preprocessors other than *eqn* and *tbl* must have a somewhat better than average understanding of how the system works. (The average required understanding is, by design, near zero.) For example someone with the following in a *troff* input file could probably also be assumed to understand the sequence of operations done by the distributed printing system *troff* scripts:

```
.\ " this has to be run through soelim before refer
.
.
.so /u0/csam/johnston/util/troff.macros
'so /u3/graphics/biblio/refer.me      \" prevent soelim from expanding this
.
.
```

The same applies to the use of *pic* and *ideal*.

The mechanism for handling T_EX input files will be somewhat different than the way the distributed printing system deals with *troff* files. The reasons are that 1) T_EX produces more meaningful error messages than does *troff*, and 2) T_EX provides an indirect referencing scheme that may require two or three passes over the input file in order to resolve the references. Neither of these are impossible to deal with in a batch environment. *Lpd* will mail messages back to the user, and *errorout* message analysis on the server could result in automatically invoking T_EX multiple times. We have not done this yet, but probably will as T_EX use increases owing to people from the VMS environment becoming familiar with the UNIX aspects of the distributed printing system.

2.4. Input to the Distributed Printing System

There are several file formats the printing system must be able to process. These formats are *troff* and *ditroff* input files, T_EX *dvi*, Tektronix, UNIX *plot*, ASCII and device specific code.

2.4.1. troff

The use of *troff* is well established, but not universal at LBL. The four UNIX systems operated by the central facility are used primarily for *troff* based text processing,

the Technical Information Department uses *troff* to drive its phototypesetter, and the Computer Science Research Department text processing is mostly *troff* based.

Ditroff is considered separate from *troff* because of the unfortunate circumstance of having a different set of available fonts. Most of the laser printer output is to Imagen printers, and the Imagen support for *ditroff* supplies a less sophisticated set of fonts than for *troff*. *Ditroff*, however, provides the features of landscape mode formatting, and a mono-spaced font. These are the main reasons for its preference over *troff*. The lack of some fonts, and diversity of other fonts create the one of the biggest nuisance factors in trying to provide reasonable typesetting services. There are now professionally designed fonts becoming available. These will help, but as yet they lack the diversity to replace the existing ad hoc fonts.

2.4.2. T_EX

The T_EX user community at LBL is smaller than the *troff* community, and is primarily VMS based. At the moment, the distributed printing system deals only with T_EX *dvi* (device "independent" intermediate) file. The main issue in supporting *dvi* output is one of accumulating the fonts on the server that are supported by the various flavors of T_EX on the user systems. The only problem in serving the T_EX community is the unfortunate circumstance that T_EX output device drivers can put in page margins unknown to the T_EX formatter. This is done in some T_EX environments and not others, so, naturally, both cases show up on our distributed printing system.

2.4.3. Tektronix

For historical reasons, the Tektronix 401x, Plot-10/TCS graphics format, like the Calcomp subroutine library interface, is ubiquitous and must be supported. (Parenthetically, the Tektronix format is not a bad choice for laser printers, being compact and providing most of the required functionality for graphics, except for a line width attribute.) The support for Tektronix files takes two forms. One is a filter that converts Tektronix code to UNIX *plot* format and then to device code. The second method is that most laser printers (including Imagens) do various degrees of emulation for Tektronix code. The Imagen emulator is reliable, but originally emulated a Tektronix 4010, a relatively low resolution device (750 × 1000). A 4014 emulator, which we have not tried, has recently become available. This new emulator emulates the 4014 w/enhanced graphics option (3000 × 4000), which is a much better resolution match with a 300 dot per inch (dpi) laser printer.

At LBL the largest source of Tektronix code is from the Cray XMP, where it is produced as a graphics metafile by most of the graphics subroutine libraries on the Cray. These files are sent to the central facility by an FTP-like program, and printed using either the distributed printing system or the central facility, Talaris-2400 laser printers.

2.4.4. Others

UNIX *plot* code is dealt with by a *plot*-to-imPRESS filter, and then handed off to the distributed printing system.

Output device specific code (e.g. imPRESS for the Imagen, QUIC for the Talaris, and Postscript for the LaserWriter) is typically generated by graphics packages. For example, the largest sources of imPRESS code at LBL (ignoring the text formatters) are ISSCO's TELL-A-GRAF, and Tektronix's GKS systems. Like most such commercial systems, they have device drivers for each specific output device. The result is that a moderate amount of device code is dealt with by the distributed printing system.

There are some ASCII files which are printed in line printer mode.

2.5. Output Devices

This section provides a description of output devices that participate in typesetting and graphics output.

The acquisition of hardware is sometimes based on rational decisions, and sometimes not. Especially with new types of hardware the wisdom or foolishness of a particular decision may not be realized until long after the fact. When we specified the first devices that would support the distributed printing system, we had three hard requirements. The device should have:

- 1) A minimum resolution of 300 dpi on the paper;
- 2) Memory to do full page bit maps, and;
- 3) An interface to Ethernet, using TCP/IP.

Subsidiary constraints were that the marking engine quality should be commensurate with the resolution, and that the print speed should be of approximately 10 pages per minute. (This last requirement is based on wanting a reasonable replacement for a Versatec, V-80 doing 5,000 pages a month.) It is worth commenting here that we were completely aware of differing needs for output "resolution". Our goal was to procure a relatively high resolution output device compared to those in use by computer users at the time. It was not our goal to convince professional graphics artists and typesetters that they should abandon their old 2,000+ dpi devices and use our new "high" resolution devices.

2.5.1. The Imagen 8/300 Printers

The requirements mentioned above resulted in an initial acquisition of one Imagen 8/300 printer with an Ethernet interface, followed six months later by two more. These devices are 300 dpi, eight page/minute, Xerographic laser printers. Conceptually, the architecture of the printer controller consists of a communication handler, a command processor, and five translators or emulators. The input file format consists of a control line giving various state information and the language of the following data. The printer gets the command line first, sets its state and invokes the appropriate translator for the remainder of the file. The translator converts the data file from one of several formats (imPRESS, line printer, Tektronix, Diablo 630 or raster image) to a page image bitmap. The bitmap scan line modulates a laser to produce a latent image on an electrostatically charged, selenium drum. This latent image is "developed" by applying a dry powder toner (usually black) to the drum. The toner adheres to the electrostatically represented latent image, and is subsequently transferred to plain paper where it is heat fused in place to produce the final output. The software architecture mentioned above is implemented on a Motorola 68000, disk based operating system using about 1.8Mbytes of memory, of which 1.1 MBytes are used for the bitmap. The processing in the controller can exhibit a fair amount of parallelism in its functioning (communication, translation and output). While this description is nominally of an Imagen printer, it applies to most of the currently available laser printers used for computer output.

The Imagen printers are the mainstay of the distributed printing system. There are about twenty printers scattered about the site, and the six original public printers each turn out 15,000 to 30,000 pages per month, or about 10 times the suggested duty cycle. Although the original requirement was for Ethernet connected printers, there are now about an equal number of serial line connected printers. The serial line printers are substantially less expensive than the Ethernet printers, are almost as fast (see "Operational

Issues", below), and can be located in places where the Ethernet does not reach.

2.5.2. The Talaris 2400 Printers and the Issue of Duty Cycle

A second type of output device was acquired primarily to support lineprinter and graphics output from the central VMS cluster, though these printers are also used by the distributed printing system. Based on our experience with the small laser printers it was decided to use similar devices on the cluster machines instead of impact printers. The requirement was to support 250,000 pages a month of lineprinter output and an additional 5,000-10,000 pages a month of graphics output, which can be done by a 75 page/minute output device running during prime time. In what may be shown to be folly instead of wisdom, we required that the needed 75 pages/minute should be provided by at least two printers for the sake of redundancy, and at most three printers since VMS could not feed multiple printers from one queue (at that time). The result of this was to purchase three 24 page/minute, Talaris printers.

The Talaris 2400 printers are based on a Xerox, XP-24 print engine. This print engine is rated at about 30,000 to 50,000 pages a month, though this is not immediately obvious until you look at the per page maintenance charges. What may prove to be our undoing on these printers is that the aggregate duty cycle for the three 24 page/minute printers is much less than for a single 75 page/minute printer like the Xerox 8700 (which has a duty cycle in excess of 500,000 pages/month). Not only is the aggregate duty cycle too low, but when one printer is down, the other two pick up the load, thereby further exceeding their duty cycle. While these XP-24 based printers (this same print engine is used by Imagen for their 24/300 printer, and by Xerox in their 3700) have settled down, in the first several months each one was down about 50% of the time owing to mechanical problems. Things got so bad at one point that the service person (who was coming in several times a week) refused to service the printers because we had put 30,000 pages through one printer in less than a week. Things have settled down, and the printers have been working well for six or eight months now. We are, however, paying substantially more than anticipated for the maintenance required to support the high duty cycle.

Beyond the print engine problems, the controllers and supporting software of Talaris does a reasonable job, and do not exhibit any problems out of the ordinary. The principle complaint with these controllers is the QMS QUIC language code. (Talaris is a software house that OEMs the QMS systems.) QUIC is the native language of the QMS controller. The problem with QUIC is that it uses a "human readable", ASCII format, and is therefore not compact. (PostScript, for all its good features is even worse in this regard.) By way of comparison, Tektronix format files typically undergo a five times expansion even when using a clever conversion algorithm to QUIC code. This expansion is a significant factor when the graphics print job has 200 complex frames that occupy 10-15 MBytes in Tektronix format, a not uncommon situation in super-computer output. Fortunately Talaris now provides a Tektronix 4014 emulator for its controller, and this file expansion is unnecessary.

These printers primarily serve the VMS systems. They coexist with the UNIX/Eunice print spoolers by having the final *lpd* output filter place the file into the VMS print queue, where it is printed by VMS like any other job.

2.5.3. Apple LaserWriters and PostScript

The time of PostScript is comming. Religious arguments aside, the main advantages of PostScript are twofold. First, PostScript provides a measure of typesetting device independence that we have not come close to before. Second, PostScript, together

with more and more powerful personal workstations, is producing a boom in affordable page composition systems (e.g. MacIntosh and IBM PC). Already people at LBL format special documents (e.g., brochures and signs) that would be difficult, if not impossible for even an experienced *troff* user. They debug these using a LaserWriter, and get final copy by taking their MacIntosh diskette to the local commercial copy shop to use the 2500 dpi Linotronic-101 phototypesetter, which is a PostScript printer connected to a MacIntosh, like the LaserWriter.

In the more traditional context of the system described in this paper, PostScript printers offer a substantial advantage in price (partly owing to discounts given the University of California) and device independence. A modern PostScript phototypesetter could be connected to the distributed printing system *just like any other user printer*, eliminating the pain and delays of having to go through an in-house (traditional) printing department.

Apple LaserWriters and local AppleTalk networks exist in plenty, and this community wants to include their printer in the distributed printing system without impairing its use by the local MacIntoshs. We have taken two approaches to this. The people who gave us PostScript (Adobe Systems) have also given us TranScript software that outputs *troff* to PostScript printers. (TeX output filters are also available.) The connection to the distributed printing system is made by sending the PostScript *troff* file to a MacIntosh over a serial line. The MacIntosh is connected to an AppleTalk network and thus to a LaserWriter. There is a spooling program in the MacIntosh (which is dedicated to this function) which sends the PostScript code from the distributed printing system to the LaserWriter. Another approach to this connection is being done by developing an AppleTalk network to Ethernet gateway based on a Sun with an AppleTalk network interface board.

2.5.4. Electrostatic Printer/Plotters

The venerable Versatec, V-80's (200 dpi, wet toner, electrostatic printer-plotters) remain attached to several systems. These fan-fold paper devices are useful as a line-printer (the laser printers do line printing no faster than they typeset) when the output listing may be 100+ pages and you do not want to incur the wrath of other printer users. The same is true for long *vgrind* output. The extensive use of *troff* on Versatec output devices is one thing that prompted the development of the distributed printing system. We found that 20-30% of our VAX 780, UNIX system cpu time was used for *troff* and its output filters for a Versatec like device. (This class of devices requires the rasterizing be done in the host system to supply the printers with bitmap images.) We immediately gained back this 20-30% of our user systems when the distributed printing system came on line, since most of the formatter processing was moved to a dedicated back end system.

2.6. Systems

This section describes the salient features of each of the "directly connected" systems, and comments on some aspects of integrating the Cray XMP.

The distributed printing system consists of server systems that drive the output devices, and client systems where input files originate. Directly connected client systems are those that have Internet access (TCP/IP on LBL Ethernet or ARPANET), and that run an *lpd* like file sender. Indirectly connected client systems handle input to the distributed printing system via file upload to a directly connected system. Output to indirectly connected server systems may be somewhat more automated. One case is described below.

2.6.1. The Server Systems

The client/server distinction among computer systems is not rigid, and those systems with unique hardware interfaces act as servers for their hardware, though in general the goal has been to affect a separation. The primary server systems are dedicated back end systems that normally do not have any users except for an occasional operator.

The servers for the distributed printing system consist of three Integrated Solutions, 4.2bsd UNIX systems. There are two 68020, VME bus systems supporting most of the printers, and one 68010, Q-bus system used mostly for development and debugging. The two 68020 systems are configured with 16 serial ports, 8 Mbytes of memory, an Ethernet interface and two 300 Mbyte disks. The serial ports support DMA output, making them suitable for attaching printers. The Q-bus system is similar but with less of everything and the addition of a 1/4" tape drive. The tape drive is used mainly for installing operating system updates. Most other tape access is via *rtar* or *rdump* using a tape drive on a VAX UNIX system.

The Integrated Solutions machines were chosen because of excellent cost/performance characteristics. These systems have proven reliable enough that we have not yet regretted the lack of a maintenance contract.

The two 68020 systems easily handle the nine public printers, and the dozen or so private printers. They will probably support almost that many again.

2.6.2. The Client Systems

The client operating systems include 4.2bsd UNIX, System V UNIX and VMS.

The 4.2bsd client systems work in the obvious way, via *lpr*. There are 8-10 such clients, including the central facility systems, several research systems, and several SUN workstations.

There are another 8-10 VMS clients that act as clients by virtue of running the Wollongong UNIX emulator.

The ELXSI system has a System V UNIX emulator, networking code from a third party, and a local version of *lpr* and *lpd*. This system can operate as a standard client, but most of its printing is done through a directly attached Imagen. The high speed of the ELXSI cpu significantly reduces *troff* turnaround, a definite advantage to users when the system has cpu to spare.

The Cray XMP is an indirectly connected client. There is an effort underway to establish an interprocess communication mechanism between the Cray and the central facility systems. Once this is done, it will not be too difficult to queue files from the Cray directly to the distributed printing system by developing an *lpd* look alike.

2.6.3. VMS Implementation Issues

Considerable effort has been put into the Wollongong UNIX emulator (Eunice) to support *lpr* and the DEC DEUNA Ethernet interface shared with DECNET. The result is that VMS systems now constitute a significant portion of the clients with much of their usage being *TeX*, graphics and some *troff*.

Eunice does not support all of 4.2bsd UNIX. *Lpr* was rewritten so not to use *select* (which did not exist when this was done), and not to use the *AF_UNIX* communication domain. The use of *syslog* was changed so not to require a daemon process. The use of *flock* was eliminated because it is redundant under VMS. To permit the use of VMS as a server system, an interface to the VMS print symbiont was implemented using the VMS *SNDJBC* function. Several new codes were added to */etc/printcap* to support the

VMS symbiont interface.

In general, UNIX style shell scripts do not do well under VMS. This is due in part to the higher overhead of starting up processes in VMS. The standard user interface functions of the distributed printing system that are implemented as scripts and invoke many processes, were rewritten to collapse the operation into one process.

There are some VMS systems that have only the UNIX networking code from Wolongong (not the full Eunice), and we have been experimenting with non-*lpd* originators. These systems have code that generates the control file, and makes *lpd* like connections to a server system.

2.7. Networks and Interconnections

Connections to output devices includes Ethernet, serial line, and parallel interface. The connections are determined by cost, location and network availability. The core of the distributed printing system is Ethernet based laser printers.

Internet connected client systems have access to the distributed printing system via TCP/IP on Ethernet, or ARPANET, or TCP/IP over point to point DECNET links using DBRIDGE. (DBRIDGE provides a mechanism for tunneling IP packets through a DECNET connection. Initially this was important, but is somewhat less so now that most systems have IP connections to the Ethernet. See reference [2].) Although not directly related, it is worth briefly describing the LBL Ethernet based internet, since this was the key to the rapid success and spread of the distributed printing system owing to it's relatively high bandwidth and wide availability throughout the site.

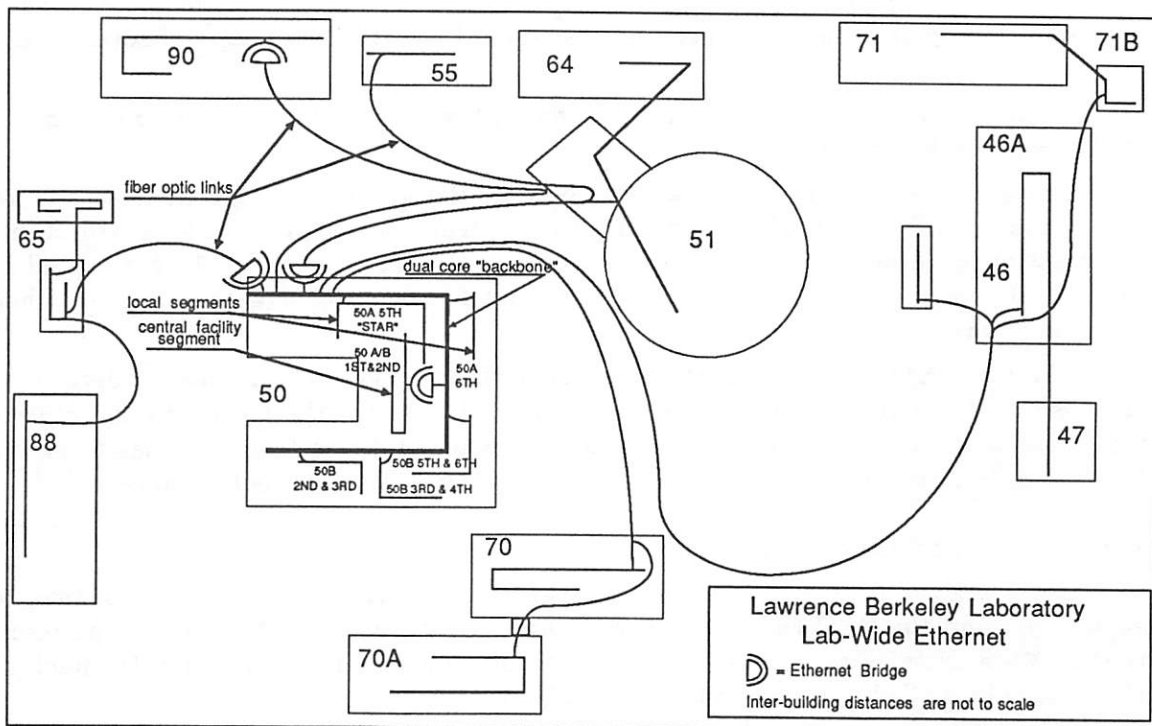


Figure 2

The LBL Ethernet consists of 2.5 Km of coaxial cable and multi-strand fiber optic links. The system is organized into core segments that are interconnected by DEC, LAN Bridge-100, protocol insensitive bridges. These bridges serve the important function of isolating the core segments by virtue of selective packet forwarding. The bridges

dynamically build destination tables indicating which packet destinations are on the "other side" of the bridge, and then only forward packets to the next core segment when the destination is located there. This is done at 15,000 packets/second, nearly the theoretical Ethernet bandwidth. These bridges are strategically located to isolate core segments with locally heavy traffic. They also serve as a "fire door" so that when one host or interface goes berserk it does not bring down the whole net.

The core segments (typically one per building) are connected to local segments (typically one per floor) via repeaters. The repeaters are used to electrically isolate the segments. They also reform the packets, but only at the signal level. As far as the Ethernet is concerned these are passive devices. See figure 2.

This large, contiguous Ethernet has several hundred attached systems running at least five different protocol suites: NSP (DECNET), TCP/IP, XNS, 3COM-XNS, and Intel, so far as we know (there could be others). Particularly as the local segments turn to "thin" Ethernet, the control of what gets attached becomes minimal.

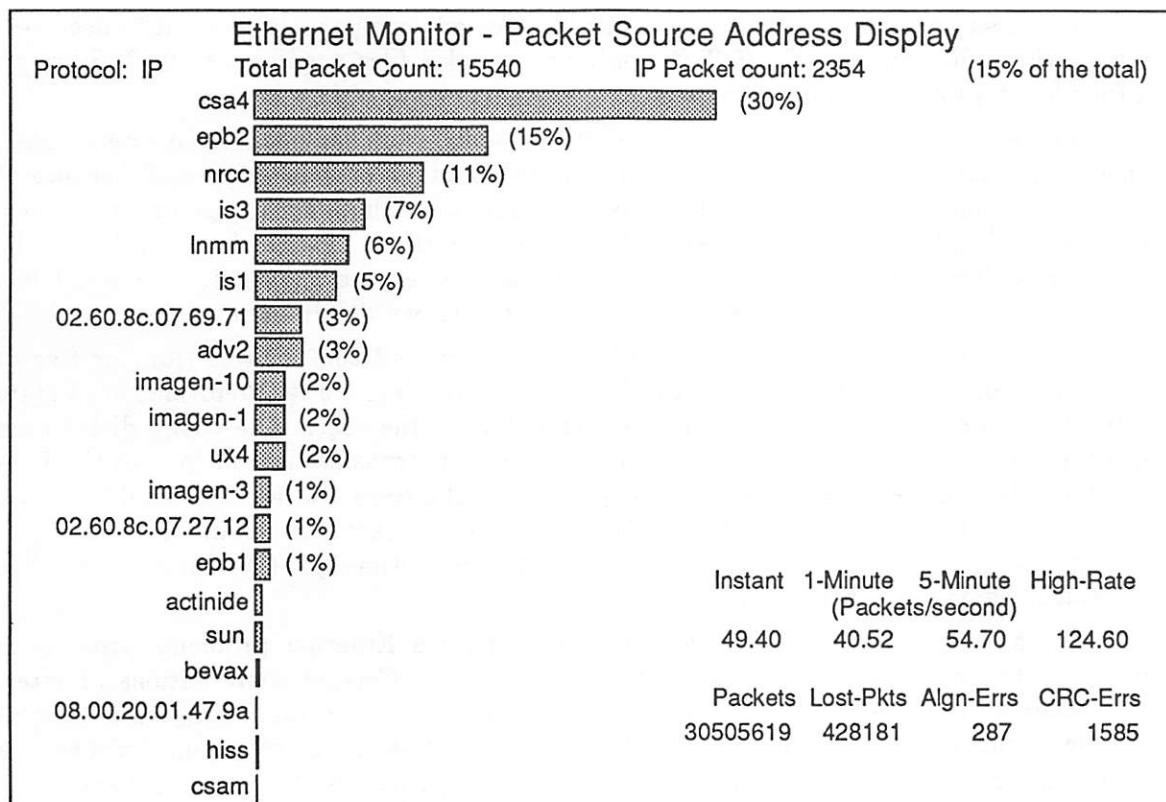


Figure 3

Figure 3 shows output from an Ethernet monitor connected to the central facility segment. The display shows sources of IP packets only, which were about 15% of the total at the time of the sample. Most of the rest of the traffic is from DECNET hosts. The IS1, IS3 and ADV2 hosts are distributed printing system server systems and Imagen-1, etc. are printers. As may be seen, printing accounts for about 20% of the IP traffic, or 3% of the total network traffic. Since the monitor is connected to the central facility segment which is connected by a bridge to the rest of the Ethernet, the graph shows only hosts on that segment, or hosts whose traffic destination is on that segment. The systems represented by their Ethernet address are of unknown identity.

3. Performance Issues

In acquiring the initial components of the distributed printing system we estimated that one 68010 system would support two or three Imagen printers running continuously. After assembling the first instantiation of the system we ran some cpu versus time-to-print-a-job benchmarks. We found that for a specific job running one, two and three simultaneous instances (job= *tbl* | *eqn* | *troff* | *catimp* | *ies*, as discussed above) that the times to complete each job were about 1.0, 1.1, and 1.5 times one job, respectively. About 90% cpu utilization occurred with two jobs running. This benchmark has not been repeated for the currently used 68020 systems, but the one job case runs a little better than three times as fast. The other figures should scale similarly since I/O utilization is small compared to cpu, and there is adequate memory to run ten or twelve simultaneous jobs.

Other benchmarks show the overhead of unnecessarily running *eqn* and *tbl* on every *troff* job is less than 5%. We did find in early tests that we got a noticeable improvement in throughput by moving the font files off the system disk to somewhat balance disk access.

The disk space requirements of a print server are comparable to a multi-user system of the same cpu size. *Troff*, *TeX*, fonts and spooled files need about 100 MBytes on a four to six printer server system.

The speed of Ethernet printers and 9600 baud serial line printers is comparable, once a job starts printing. Ethernet printers are about 10-15% faster overall because of font cache reloading. The Imagen printers will cache fonts, but not a huge number. Since *troff*, *ditroff* and *TeX* all use different fonts, each of those jobs tend to invalidate the font cache. The reloading seems to be much faster over the Ethernet. The serial line printers now run at 19,200 baud, which somewhat narrows the difference.

Initially we had substantial concerns that running all the printers (four or five of which run almost continuously) on the Ethernet would degrade its performance. Figure 3 in the network section shows that this has not been the case. The entire distributed printing system, including the user to server system file transfers and output to the Ethernet printers, is never more than a few percent of the total traffic. The total Ethernet traffic is almost never more than 10-15% of the 1,000 packet per second that we feel is the practical maximum for the Ethernet to function optimally. (We have no diskless workstations on the central Ethernet at this point.)

We have had some trouble with hardware related Ethernet problems, from both interface failures and incompatible Ethernet hardware. Certain combinations of interface boards, cable transceivers, and repeaters work well and certain ones don't work at all. Detecting this type of problem is difficult, and isolating the offending hardware is even harder. It requires perseverance from the technicians and systems programmers responsible for operations.

Since UNIX pipes are an expression of (course grained) parallelism, this application should do well on a multi-processor system like the Sequent, Balance 8000 (where we would love to try it). These machines schedule processes from a single run queue to the lowest priority processor, and handle interrupts in special hardware. The distributed printing system job (*tbl* | *eqn* | *troff* | *catimp* | *ies*) would naturally distribute itself over many processors.

4. Suitability of *lpd*

This section contains comments on changes that were made to *lpd*.

An accounting mechanism is now being added. Each control (*cf**) file must provide a valid account number that is recorded, together with page counts, by job. Following the philosophy that simple is best, charging is done only on a per printed page basis.

There were only two types of changes made to *lpd*. The first change was to increase the number of entries in the control file. These changes were made to accommodate accounting information and filter arguments, such as landscape mode flags and hard margin changes.

The second change to *lpd* was to prevent it from attempting to restart a job. The scripts that implement the distributed printing system transform the control file in a way that is not idempotent, and a second pass results in nonsense. Even without this circumstance, jobs of the type being discussed here almost never fail in a recoverable way and restarting just results in *lpd* looping. Many debugging facilities were also added to *lpd*.

5. Operational Aspects

Once the system is running, you have to find out if operators can run it, or, better yet, if it can run itself. The goal of this system is to have the printers be self service (user operated) up to, but not including, preventive maintenance, and not to require operators. Once the "sanity" checking was added (as described below) we approximated this.

A key to reducing operator participation (i.e. taking phone calls from irate users who know that the distributed printing system is running, but nothing has come out of the printer for an hour) was to add simple sanity checks to make sure that the *tbl* | *eqn* | *troff* | *output_filter* | *communication_filter* chain had not become wedged, and if it had, to notify someone automatically. At the moment this is done via the simple expedient of having a *crontab* triggered process check the printer status file to make sure that it changes periodically. If it does not, a message is sent to a central facility operator who can check the job manually. The rule of thumb that we use for *troff* jobs is that the cpu time (on a 68010 system) should be about one minute per five kilobytes of input file. Anything much beyond this probably means that *troff*, or its preprocessors, is looping.

Initially there was a fair amount of educating the users. Users would look at the job queue (via *lpq*), observe that their job was active, and the printer was idle. What they failed to realize is that "active" means that *troff* is working on formatting the input, not that output is yet being sent to the printer. (The job pipe takes some time to fill up.) Also, users are now educated on how to add paper, change toner, clear jams, etc., thanks to the good design of the Canon print engine.

Operational tools have been provided to permit moving printers from one server to another, changing user systems to server systems, etc. These are described in reference [1].

The Canon print engine used in the Imagen 8/300 printer is rated at approximately 3,000 pages a month and a total lifetime of about 100,000 pages. From the beginning we have consistently put more than 5,000 pages a week through the public printers. The company that maintains the Imagens for us refused to maintain the print engines after discovering that the internal page counter on one printer was over 500,000. This turns out not to be a problem for two reasons. First, the Canon engines are reliable and a small amount of preventive maintenance done by the operators will keep them operating for months at a time. Secondly, as the printers age past their advertised lifetime, what usually goes wrong is that the lubrication on some of the internal working parts fails and they start squeaking and sometimes binding. Since we have more than fifty Canon print engines (LaserWriters, Laser Jets, Imagens, etc.) the local maintenance group had one

person trained in their maintenance. The squeaking and binding problem takes them a few hours to correct. The replacement Canon engines are about \$1,000, but we have yet to replace one. As pointed out elsewhere, these conservative duty cycle ratings do not manifest themselves on faster printers, whose advertised duty cycle is much closer to reality.

In the issue of priority queues (or how do you keep the Director's secretary, who has just queued an important one page letter, from killing the graduate student who queued his 100 page thesis just before that), the solution has been mostly one of education. However we do impose software implemented page limits on some of the public printers. The best solution, though, is to buy more printers.

6. Acknowledgements

The authors were the primary architects and implementors of the distributed printing system and assisted in the transition from prototype to a production system. Maintenance, and many refinements, are carried out by Bob Rendler of the Computer Center. Marty Gelbaum and Wayne Graves were responsible for making the VMS end of things work with Eunice. Theresa Breckon wrote the MacIntosh spooler to get output to AppleTalk network connected LaserWriters, and Van Jacobson has been experimenting with Ethernet-AppleTalk gateways to this same end.

The quite reasonable UNIX software of Imagen made the initial effort manageable, and thanks goes to Dan Curtis of Imagen for his expert assistance with that software. The similarly good Adobe Systems, Transcript software made possible the attachment of the LaserWriters.

7. Bibliography

[1]

"An Administrator's Guide to the LBL Distributed Printing System,"
Johnston, W. E., LBL Report, PUB-3055, Lawrence Berkeley Laboratory,
Berkeley (1986).

[2]

"4bsd UNIX TCP/IP and VMS DECNET: Experience in Negotiating a Peaceful
Coexistence,"
Jacobson, Van, Craig Leres, Joseph Sventek, and Wayne Graves, *USENIX
Summer Conference Proceedings*, (1984).



